

# Visualisation de matchs de football robotique

*Lucie CHASAN*

*Florian ESCURE*

*Lucie MATHÉ*

*Elliott MONTERO*

4 avril 2019



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Besoins</b>	<b>5</b>
2.1	Fonctionnels . . . . .	5
2.1.1	Priorité 1 . . . . .	5
2.1.2	Priorité 2 . . . . .	7
2.1.3	Priorité 3 . . . . .	8
2.2	Besoins non fonctionnels . . . . .	10
<b>3</b>	<b>Architecture du projet</b>	<b>11</b>
3.1	Code fourni par le client . . . . .	11
3.1.1	hl_communication . . . . .	11
3.1.2	hl_monitoring . . . . .	13
3.2	Architecture du code . . . . .	14
3.3	Fonctionnement du projet . . . . .	16
3.3.1	L'initialisation . . . . .	16
3.3.2	La récupération d'un message . . . . .	17
3.3.3	La récupération d'une image . . . . .	17
3.3.4	L'annotation sur l'image . . . . .	18
3.4	Avantages de l'architecture . . . . .	18
3.4.1	Ajout d'annotations . . . . .	18
3.4.2	Ajout d'équipe et de robots . . . . .	19
<b>4</b>	<b>Fonctionnalité implémentées</b>	<b>20</b>
4.1	Les annotations . . . . .	20
4.1.1	Les couleurs . . . . .	20
4.1.2	La position . . . . .	21
4.1.3	La direction . . . . .	21
4.1.4	La trace . . . . .	22
4.1.5	La balle . . . . .	22
4.1.6	La position souhaitée (Target) . . . . .	23

4.1.7	Les lignes du terrain et le score . . . . .	24
4.1.8	Délai et optimisation des annotations . . . . .	24
4.2	Les outils de annotateImage . . . . .	25
4.2.1	Init_match . . . . .	25
4.2.2	Main_annotationImage . . . . .	26
4.3	L'interface . . . . .	26
4.3.1	Choix de l'API . . . . .	26
4.3.2	Affichage de la vidéo . . . . .	27
4.3.3	Les fonctionnalités de l'interface . . . . .	27
4.3.4	Le slider . . . . .	29
<b>5</b>	<b>Tests effectués</b>	<b>30</b>
5.1	Tests sur l'utilisation du Json . . . . .	30
5.2	Tests sur la lecture des logs . . . . .	31
5.3	Tests sur la performance . . . . .	32
5.3.1	Fichiers de test . . . . .	32
5.3.2	Paramètres des tests . . . . .	33
5.3.3	Test de vitesse d'annotation . . . . .	33
5.3.4	Performance selon le codec vidéo . . . . .	33
5.3.5	Impact des blend . . . . .	35
5.3.6	Limites de puissance . . . . .	36
<b>6</b>	<b>Problèmes rencontrés et limites</b>	<b>39</b>
6.1	Les vidéos à annoter . . . . .	39
6.1.1	Le live . . . . .	39
6.1.2	Taille des vidéos . . . . .	39
6.1.3	Format des vidéos . . . . .	40
6.2	La transparence . . . . .	40
6.3	Le slider . . . . .	41
6.4	La trace . . . . .	41
<b>7</b>	<b>Conclusion</b>	<b>43</b>
	<b>Bibliographie</b>	<b>45</b>
<b>A</b>	<b>Architecture des messages</b>	<b>46</b>

# Chapitre 1

## Introduction

La robotique est un domaine en plein essor. Depuis que le superordinateur Deep Blue a battu Kasparov aux échecs en 1997, le monde s'est rendu compte du potentiel de l'intelligence artificielle.

Au-delà de la seule intelligence, le sport nécessite des capacités physiques. Aussi, lors d'un match de football, l'environnement est dynamique, il évolue sans cesse. Un robot qui joue au football se voit donc davantage confronté à la complexité du monde réel.

La Robocup est une compétition où des robots autonomes s'affrontent sur plusieurs disciplines. Depuis 1997, le tournoi de football robotique est une discipline majeure du tournoi. Comme l'intelligence artificielle a surpassé l'homme aux échecs et au jeu de Go, le but dans cette compétition est de créer une équipe robotique capable de surpasser une équipe humaine.

Il existe d'autres matchs et tournois en dehors de cette compétition, mais une victoire à la Robocup reste un enjeu majeur pour les équipes.

Notre client est l'équipe **Rhoban**, évoluant dans la compétition avec des robots humanoïdes kid-size, c'est à dire des robots d'apparence humaine mesurant entre 40 et 90cm.

Notre but ici est de créer un outil d'annotation destiné à mieux comprendre le match et les comportements des robots. À partir des messages envoyés par les robots et d'un flux vidéo, nous allons donc annoter les images en fonction de ce que le robot perçoit et souhaite faire.

Nous sommes capables d'annoter la position et la direction du robot. On peut également incruster, pour un seul robot à la fois, sa perception de la

balle et l'historique de ses positions. Enfin, nous sommes capables d'afficher la position que souhaite atteindre un robot.

Nous proposons deux manières d'afficher la vidéo. Premièrement, un outil simple permettant de visualiser une vidéo annotée selon des paramètres prédéfinis par l'utilisateur.

Nous pouvons également afficher la vidéo dans une interface graphique. Dans celle-ci, on peut gérer l'avancement de la lecture grâce à un slider et changer les annotations en cours de visionnage.

On peut voir dans la Figure 1.1 une image de la vidéo non annotée telle que fournie par le client :

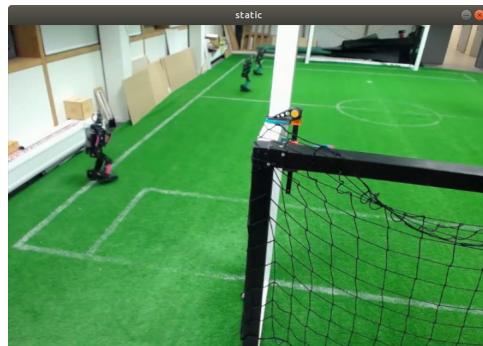


FIGURE 1.1 – Image de la vidéo avant traitement

Maintenant, on peut voir dans la Figure 1.2 une image de la vidéo avec toutes les annotations proposées (sauf les lignes du terrain, pour une meilleure visibilité). La balle (en gris à gauche), la position souhaitée du robot (la croix et les pointillés) ainsi que la trace (anciennes positions) sont celles du robot 4 de l'équipe bleue.



FIGURE 1.2 – Image de la vidéo après traitement

# Chapitre 2

## Besoins

Ce projet consiste à proposer une solution logicielle pour incruster les données d'un match de football robotique sur un flux vidéo.

Ce logiciel devra répondre aux besoins qui ont été définis au préalable avec les clients, des besoins *fonctionnels* et *non fonctionnels*.

La liste des besoins de notre cahier des besoins n'était pas bonne car notre compréhension du code fourni par le client était à l'époque très faible. Nous rencontrions des problèmes avec **Catkin** (Build system, fourni par le framework ROS spécifique pour l'écriture de logiciel de robot), ce qui nous empêchait de compiler. Ces problèmes ont rapidement été surmontés, ce qui nous a permis de mieux cerner le projet.

### 2.1 Fonctionnels

Comme tous les besoins n'ont pas une importance similaire, nous avons décidé de les séparer en 3 niveaux de priorité.

#### 2.1.1 Priorité 1

Les besoins fonctionnels de priorité 1 sont indispensables au fonctionnement du programme.

**Interface** : Les clients désiraient une interface dans laquelle serait affichée une vidéo annotée. Depuis cette interface, il doit être possible de modifier les choix d'annotations.

Nous avons donc créé une interface en utilisant le framework **Qt**. Elle doit communiquer avec le package qui s'occupe d'incruster les annotations sur la vidéo. Elle peut ainsi récupérer les images annotées et les afficher.

L'interface affiche aussi des informations utiles (le temps et le score) pour le match en cours. Elle propose des moyens d'interaction pour personnaliser les annotations (menu Annotation Choice). On peut contrôler le visionnage de la vidéo grâce aux boutons Pause, Fast Forward et au Slider.

**Annotation de la position et de la direction** : Certaines annotations sont indispensables pour comprendre les actions des robots. Il est important d'incruster sur la vidéo l'endroit où chaque robot pense être, ainsi que sa direction.

Nous avons choisi de représenter la position par un point de taille fixe de la couleur de l'équipe du robot (une couleur par équipe). La direction est une flèche de même couleur que le point qui part du centre du point position et dans la direction indiquée par les données du robot.

Nous avons implémenté ces solutions en utilisant les fonction de dessin d'**OpenCV**, qui ne demandent que très peu de puissance de calcul 5.3.3.

**Compréhension et appropriation du code du client (et Catkin)** : Ce fut un des gros problèmes du début de projet. Nous avons eu beaucoup de mal à compiler le code client fourni, qui était nécessaire pour l'avancée du projet (notamment pour recevoir les messages des robots et les lire ; il sert aussi à décomposer la vidéo en images et permet de transformer la position du robot en point sur l'image en prenant en compte les paramètres de la caméra). Le problème venait des dépendances avec Catkin, que nous ne savions pas comment utiliser.

Finalement, après de nombreux essais pour faire fonctionner CMake, Catkin et les autres dépendances, et avec de l'aide de nos clients, nous avons réussi à compiler le code fourni. Nous avons ainsi pu intégrer notre code dans ce système de compilation. Plus tard, nous avons passé les modules clients ainsi que le paquet Catkin en submodule de notre dépôt git pour pouvoir les mettre à jour lorsque c'est nécessaire.

Ces besoins sont la base du projet, et étaient le minimum à atteindre pour avoir un programme utilisable.

Nous avions pensé devoir calculer l'équation de plan permettant de récupérer un point sur l'image à partir d'une position sur le terrain. Ces deux articles [Kannala J., 2006] et [Matas J., 2000] nous ont offerts une première approche sur le problème.

Lorsque nous avons pu lire le code fourni par le client, nous avons vu comment utiliser ses fonctions qui prenaient déjà en compte l'équation et nous n'avons donc pas eu à travailler dessus.

### 2.1.2 Priorité 2

Ces besoins dépendent de ceux de priorité 1, mais sont indispensables au bon fonctionnement du programme.

**Relier l'annotation des images à l'interface :** L'interface offre du confort à l'utilisateur, notamment pour choisir les annotations. L'annotation de l'image se fait dans le module **annotateImage**. Pour pouvoir afficher les images annotées dans l'interface, il a fallu mettre en place une communication entre ces deux modules.

Dans la première moitié du projet, nous avons eu beaucoup de difficultés à relier ces deux parties. Ceci était principalement dû à nos (nombreux) problèmes d'architecture précédant le rendu de la V1. Notre nouvelle architecture, plus robuste, nous a permis de résoudre ce problème de communication inter-modules.

Auparavant, nous appelions un manager depuis un autre manager d'annotation. Le premier manager prenait alors la main, et ne laissait pas le second s'exécuter. Nous avons résolu ce problème avec un QTimer qui met périodiquement le manager à jour.

Nous appelons donc l'annotation d'images périodiquement pour rafraîchir l'image *augmentée* dans l'interface. Celle-ci communique avec le manager pour choisir les annotations à incruster.

**Avoir une architecture permettant une adaptabilité du projet** : Les informations envoyées par les robots peuvent changer. À l'avenir, ils enverront peut être de nouvelles informations que nous ne pouvons pas afficher actuellement (comme l'indication du périmètre de sécurité autour d'un robot pour un coup-franc par exemple!). Nous devons faire en sorte qu'il soit simple d'ajouter de nouvelles annotations au code.

Si on souhaite ajouter un nouveau type d'annotation, il suffit d'ajouter une fonction d'incrustation dans *annotation.cpp* (paquet *annotateImage*), et d'ajouter un champ pour l'annotation dans le fichier json *annotation\_settings.json* (qui gère les caractéristiques des annotations (taille, couleur, ...)).

**Permettre à l'utilisateur de personnaliser ses annotations et de faire des choix** : L'utilisateur doit être en mesure de configurer les paramètres esthétiques des annotations (taille, couleur ...). En effet, lors d'un match avec plusieurs robots, afficher toutes les annotations pour tous les robots rendrait la vidéo complètement illisible tant il y aurait de contenu à l'écran.

De plus, ne pas avoir de personnalisation des annotations rendrait le logiciel très peu flexible, et restreindrait l'éventail des utilisations possibles. C'est pourquoi nous avons mis en place un fichier *annotation\_settings.json* contenant les paramètres principaux de chaque annotation, et qui est apte à être modifié pour toute utilisation souhaitée.

### 2.1.3 Priorité 3

Ces besoins ne sont pas indispensables au fonctionnement du programme, mais lui permettent de gagner en ergonomie.

**Faciliter le déplacement dans la vidéo** : Comme dans la plupart des logiciels de lecture de vidéo, il est utile de pouvoir se déplacer dans la vidéo. Nous avons solutionné ce problème en implémentant un bouton *play/pause*. Nous avons également ajouté la possibilité de passer en mode *fast-forward* pour avancer plus rapidement, ainsi qu'un slider sous la vidéo, qui va permettre de se déplacer dans la vidéo et de visualiser son avancement.

**Permettre à l'utilisateur de personnaliser l'incrustation** : Ce besoin est un peu englobé dans le besoin de l'interface, mais n'était pas indispensable pour obtenir une interface fonctionnelle.

Il est souhaitable que l'utilisateur puisse faire des choix de manière intuitive depuis l'interface pour personnaliser son expérience. Par exemple, choisir les annotations à afficher ou non.

Nous avons implémenté de nombreuses solutions à ce besoin dans l'interface, comme le menu Annotation Choice qui permet de choisir quelles annotations on souhaite afficher, ou encore la trace de quel robot on souhaite afficher.

**Ajouter de nouvelles annotations** : L'incrustation de la position et de la direction est un besoin essentiel. Cela représente toutefois peu d'informations, et il serait souhaitable d'en proposer d'avantage.

Grâce à notre architecture, nous avons pu rajouter de nombreuses annotations comme par exemple une trace des dernières positions d'un robot, ou la position de la balle telle que perçue par un robot. On choisit le robot pour lequel afficher l'annotation car l'affichage serait trop surchargé si on l'affichait pour tous les robots.

**Afficher l'état de la partie sur l'interface** : Les clients souhaitaient voir les informations sur les équipes, les robots, et le match dans l'interface.

Nous avons choisi d'afficher le score ainsi que la liste des robots de chaque équipe, mais aussi quelle annotation est activée pour chaque robot.

## 2.2 Besoins non fonctionnels

**Dépendances à jour** : Pour faire tourner le projet, nous utilisons de nombreuses dépendances (qui sont listées dans le README d'installation) comme OpenCV 3, QT 5.9.5, mais aussi Catkin, CMake, python ou encore protobuf.

Sans ces dépendances, on ne peut pas compiler le projet (suivre le *README* pour configurer).

**Vidéo du match** : La vidéo du match doit être de résolution 640x480, les formats avi, mov et mp4 sont tous acceptés.

La caméra doit rester statique pendant tout l'enregistrement (on ne peut pas mettre à jour l'équation de plan), et ne pas avoir de trop grosse déformations (par exemple l'effet fisheye d'une caméra grand angle va générer des erreurs).

**Logs et fichiers de configuration** : La vidéo du match doit se trouver dans le même dossier que les logs du match, ainsi que les fichiers de configuration json.

Voir l'exemple avec les dossiers présents dans le google drive donné dans le *README*, qui contient les fichiers et les vidéos de matchs de football robotique.

**Equipes de robots qui respectent la hiérarchie des logs** : Tous les robots ne communiquent pas de la même manière, et tous les logs ne suivent pas la même architecture.

Nos clients ont mis en place une architecture de logs. Les logs fournis doivent donc suivre l'architecture définie par le client pour être exploitables par notre logiciel.

# Chapitre 3

## Architecture du projet

### 3.1 Code fourni par le client

Le code fourni par le client se compose de deux packages. Il nous a été spécifiquement demandé de travailler avec OpenCV3. Pour nous aider à utiliser OpenCV, nous avons utilisé [Gary Bradski, 2008].

#### 3.1.1 **hl\_communication**

Le premier package **hl\_communication** contient les classes nécessaires à la lecture des messages des robots.

Nous avons dans ce package les multiples fichiers *.proto* décrivant l'architecture des messages.

L'architecture globale des messages est disponible en annexe A, page 46. La Figure 3.1 ci-dessous nous montre une version simplifiée avec seulement les éléments des messages que nous utilisons.

Un message se compose de deux parties essentielles : les messages du **Game\_Controller** et ceux des robots. La hiérarchie de départ du message est contenue dans le fichier *wrapper.proto*.

Le **Game\_Controller GC\_Msg** contient toutes les données réelles du match, c'est à dire toutes les données envoyées par la table d'arbitrage : le temps, les équipes (**GCTeamMsg**) et leurs robots (**GCRobotMsg**).

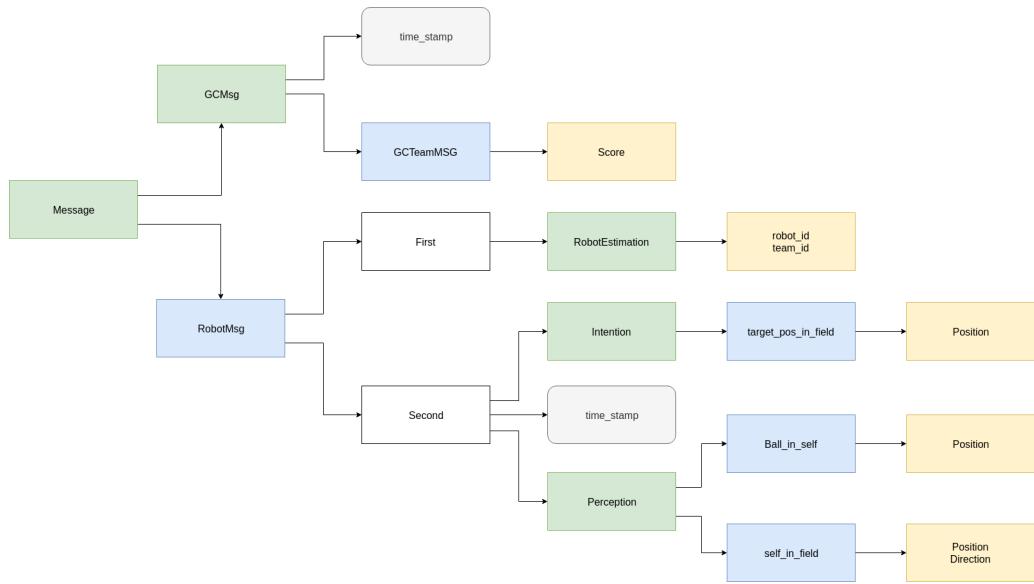


FIGURE 3.1 – Architecture des messages

*En vert, les noms se rapportant aux fichiers proto. En bleu, ce sont les sous-ensembles des messages et en jaune les éléments du match que nous récupérons.*

Dans ces messages nous récupérons le numéro des équipes et le score. Nous recevons également un message **Robot\_Msg** par robot. Dans notre programme nous traitons plusieurs éléments de ces messages :

- le **time\_stamp** pour mesurer le temps écoulé depuis l'envoi du message ;
- le **Robot\_Estimation** : qui nous donne son numéro d'équipe et de robot ;
- la **Perception** du robot : on récupère ici la position et la direction du robot (*self\_in\_field*) et de la balle (*ball\_in\_self*) ;
- Enfin, l'**Intention** du robot est utilisé pour avoir sa position souhaitée (*target\_pose\_in\_field*).

Les principales fonctions que l'on utilise de ce package sont contenues dans la classe **MessageMonitoring** dont nous utilisons principalement la fonction **getStatus(time\_stamp)** qui nous donne le message correspondant à un certains temps *time\_stamp*.

### 3.1.2 hl\_monitoring

Ce second package contient essentiellement le **MonitoringManager** qui nous permet de rythmer l'avancée de la vidéo.

Tout d'abord, il lit le fichier de configuration *match\_settings.json* puis *replay.json/live.json* qui contient le nom du binaire où l'on récupère les messages des robots, les informations sur les caméras et leur vidéos et enfin un booléen pour savoir si on est en replay ou en live.

C'est également lui qui va initialiser le **MessageManager** vu précédemment. Tant que ce manager fonctionne, il va récupérer l'image à un temps donné de la vidéo.

La classe **Field** permet d'afficher les lignes du terrain grâce à la fonction **tagLines**.

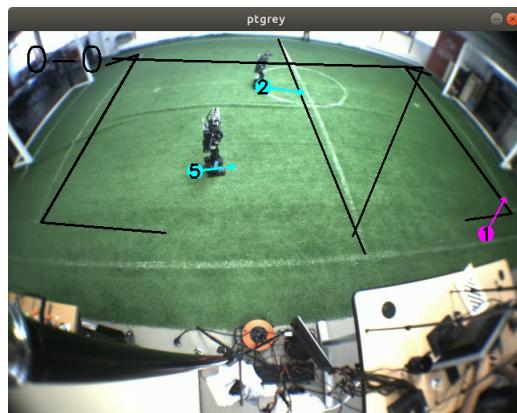


FIGURE 3.2 – Test sur l'affichage des lignes du terrain

Elle nous permet de vérifier le bon fonctionnement de la calibration de la caméra. Par exemple, dans le cas de la vidéo *ptgrey.avi* tournée le 20/02/2019, les lignes du terrains ne correspondent pas à la réalité.

Nous pouvons voir sur la photo que la ligne de la zone du gardien proche du joueur 1 en rose est mal placée. En revanche, même si la ligne est mal placée, la position du robot est bien placée par rapport à la ligne.

**Hl\_monitoring** contient toutes les fonctions de calibration de la caméra, dont l'outil *static\_calibration* (qui nous permet de calibrer une caméra une fois que l'on a récupéré ses paramètres intrinsèques). Il contient aussi la fonction **fieldToImg**, qui transforme des positions du plan réel au plan de l'image en fonction des paramètres de la caméra.

Le package **hl\_monitoring** contient aussi *basic\_monitoring*. C'est un exécutable permettant de tester tout ce qui nous a été fourni par le client. C'est dans ce fichier que nous avions fait nos premiers tests d'affichage avant d'en créer un nous-même.

## 3.2 Architecture du code

Nous avons ajoutés deux packages à ceux fournis par le client.

D'une part nous avons **annotateImage** qui nous permet d'annoter la vidéo et de l'afficher d'une façon très basique et d'autre part **Interface** qui nous permet d'afficher la vidéo avec une interface utilisateur et de changer les annotations en cours de lecture.

L'annotation d'image se compose de différentes classes :

- des classes outils telles que **Position** ou **Direction** : nous permettant de mieux gérer les éléments récupérés dans les logs ;
- les classes **RobotInformation** et **Team** : Team se compose d'une map de RobotInformation, elle est essentielle pour différencier les robots lors de la réception des messages ; RobotInformation nous permet donc de stocker le dernier message de chaque robot et d'autres informations utiles aux annotations (la trace principalement) ;
- la classe **Annotation** qui contient une fonction par annotation qu'il est possible d'ajouter.

Le fichier principal de *annotateImage* est *main\_annotationImage*, il permet de lancer la vidéo et d'annoter la vidéo grâce aux deux fichiers de configuration *match\_settings.json* et *annotation\_settings.json*. Le détail des choix possibles pour ces fichiers se fera dans la partie suivant, fonctionnalités implémentées.

Enfin, nous pouvons retrouver trois fichiers de tests que l'on détaillera dans la partie appropriée.

L'interface a été conçue à partir de l'API Qt. Elle est composée de différentes classes qui héritent de classes Qt :

- **MainWindow** : hérite de QMainWindow, représente la fenêtre principale de l'interface ;
- **TeamPanel et RobotPanel** : héritent de QWidget, ces deux classes nous permettent de faire savoir l'état d'une annotation pour chaque robot via des QLabel ;
- **ChoiceDialog** : hérite de QDialog, représente la fenêtre pop-up servant à modifier les choix d'Annotation ;
- **ChoiceComboBox** : hérite de QWidget, permet de sélectionner le numéro d'une équipe et d'un robot. Créée par choix de factorisation puisque ce mode de choix était présent 3 fois dans ChoiceDialog.

Le fichier principal *main* permet d'exécuter l'interface.

Voici donc comment se trouve notre architecture, les deux packages en haut sont ceux du client et en bas ceux que nous avons ajoutés :

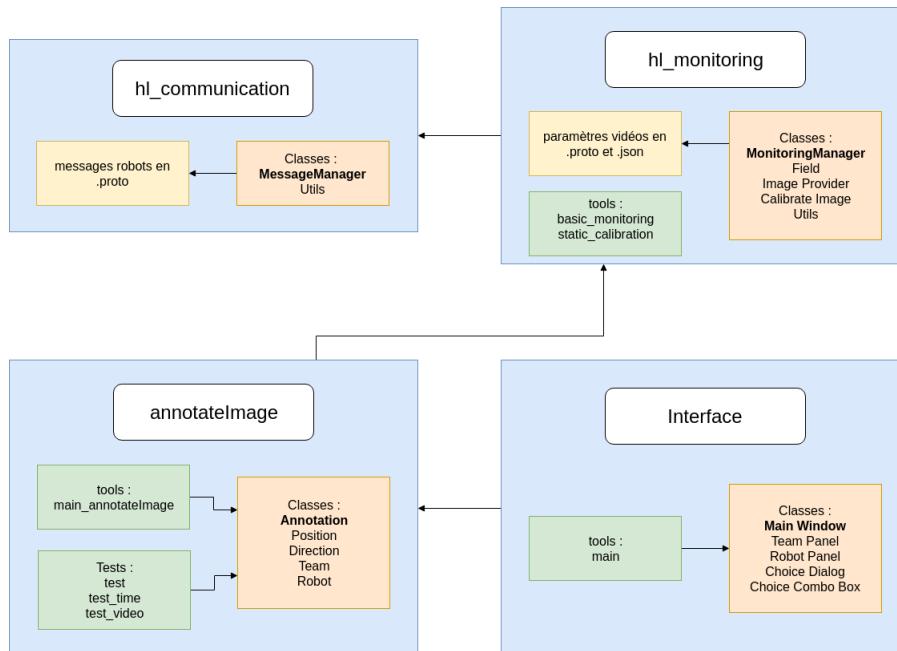


FIGURE 3.3 – Architecture des packages

## 3.3 Fonctionnement du projet

Pour expliquer le fonctionnement de notre projet, nous allons nous appuyer sur le package **annotateImage** essentiellement, sachant que l'interface fonctionne de la même façon avec les outils d'affichage en plus.

### 3.3.1 L'initialisation

Au lancement de l'exécutable de `annotateImage`, `main_annotationImage`, plusieurs fichiers sont indispensables comme nous pouvons le voir dans la Figure 3.4 ci-dessous.

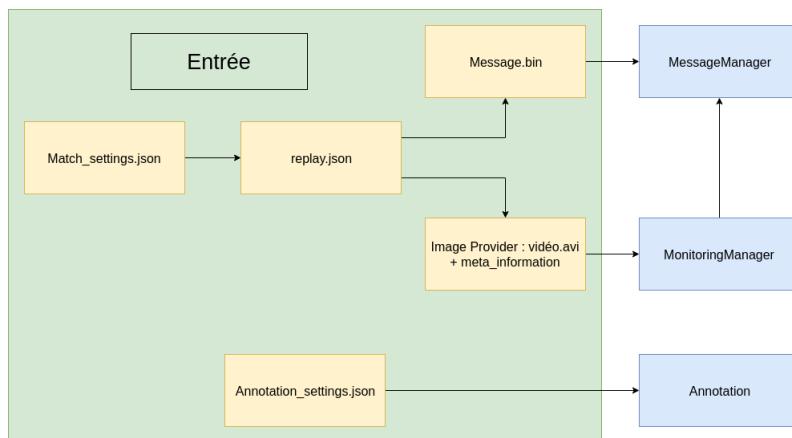


FIGURE 3.4 – Entrée et initialisation du programme

Le premier, *match\_settings* contient la configuration du projet qui va être dans un fichier *replay.json* ou *live.json*. Le fichier *replay.json* contient d'une part le nom du binaire contenant les messages permettant de créer un **MessageManager** et d'autre part le nom de la vidéo à traiter, ainsi que celui du binaire contenant les informations de la caméra associée permettant d'avoir le **MonitoringManager**.

Le second fichier indispensable est *annotation\_settings.json* qui nous permet de créer un objet **Annotation** qui lit le fichier et prend en compte tous les paramètres nécessaires à chaque annotation.

À l'initialisation, nous créons également la variable "now" qui est un *time\_stamp* et qui rythme l'avancée de la vidéo. Il correspond au *time\_stamp* de l'image en cours de traitement.

### 3.3.2 La récupération d'un message

Avant de nous occuper des images des vidéos, nous récupérons le message associé au *time\_stamp*.

Comme nous pouvons le voir dans la Figure 3.5, nous récupérons d'une part les informations de chaque **Team** (le score) et des **RobotInformation** (le Robot\_Msg). Nous accédons aux robots à travers leur **Team**.

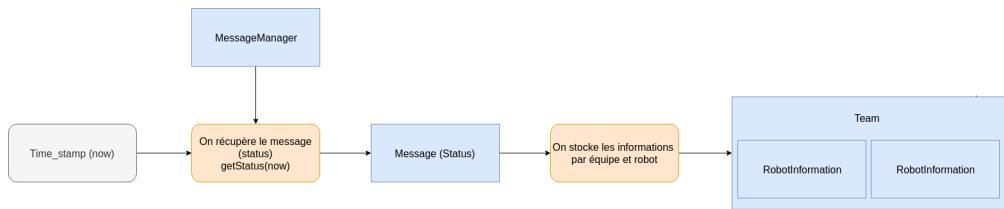


FIGURE 3.5 – Récupération des messages

### 3.3.3 La récupération d'une image

À chaque *time\_stamp*, nous allons donc récupérer l'image associé mais il est possible de lancer plusieurs vidéos en simultané. La récupération d'une image se fait donc comme détaillée Figure 3.6 :

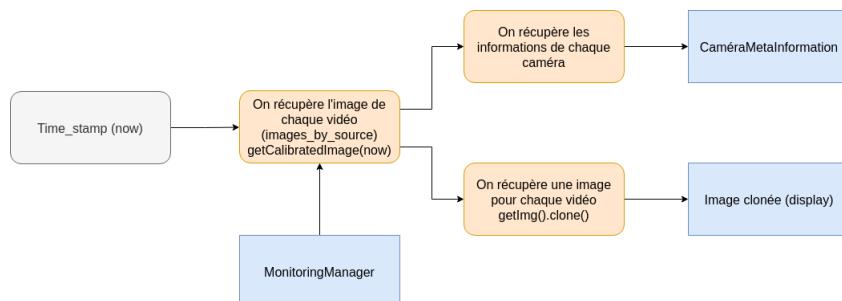


FIGURE 3.6 – Récupération des images

Nous récupérons d'abord une std::map avec une image par vidéo puis nous récupérons une image et les meta-informations de la caméra associé. L'affichage se fait donc image par image pour chaque vidéo, c'est ce qui permet d'afficher deux vidéos en simultané.

### 3.3.4 L'annotation sur l'image

Une fois que tous les éléments ont été recueillis, nous pouvons annoter l'image comme on le voit dans la Figure 4.7. Nous annotons image par image et robot par robot. Le *time\_stamp* nous permet de savoir si le dernier message du robot n'est pas trop vieux et à définir l'opacité pour l'affichage optimisé.

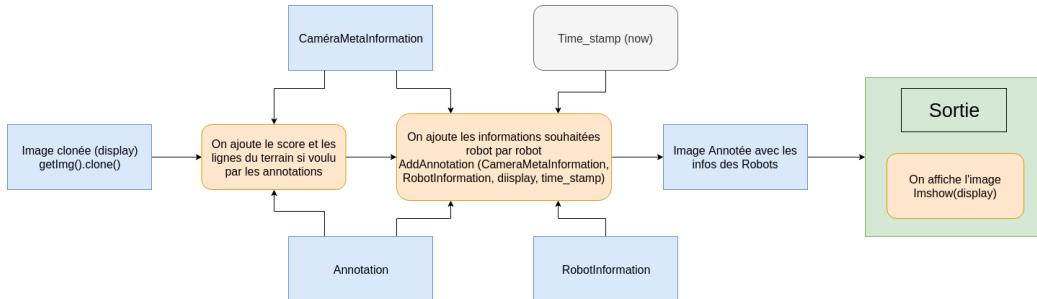


FIGURE 3.7 – Annotation des images

Notre classe d'annotation comprend la fonction **addAnnotation** globale et une fonction par annotation (par exemple : **annotatePosition**, **annotateDirection** ..).

Dans la fonction **addAnnotation**, nous décomposons le message contenu par le **RobotInformation**.

Si les éléments que nous souhaitons annoter sont présents alors nous regardons si l'utilisateur veut afficher l'annotation.

Si les deux conditions sont remplies, alors nous appelons la fonction d'annotation adéquate.

## 3.4 Avantages de l'architecture

### 3.4.1 Ajout d'annotations

Si dans le prolongement du projet nous voulons récupérer d'avantage d'informations et ajouter les annotations, notre architecture est adaptée.

Nous stockons déjà le message entier du robot (**RobotMsg**) dans la classe **RobotInformation**.

Ajouter des annotations comme la position des adversaires vu par le robot est donc assez simple. Il suffit d'ajouter une fonction **annotateOpponent** qui est similaire à l'affichage de la balle (mais avec une autre couleur et une autre taille).

Également, nous avons déjà prévu le stockage du **GCRobotMsg** contenant les penalty, les fautes et le temps restant de pénalité du robot. Il sera donc assez simple de les ajouter.

### 3.4.2 Ajout d'équipe et de robots

En raison de moyens limités, notre programme n'a été testé qu'avec deux équipes avec 2 robots contre 1. Notre architecture a été prévu pour tenir un nombre illimité de robots, ce que nous avons testé en simulant des robots (voir Figure 3.8).

Il semble également être possible d'ajouter des équipes pour la partie **annotateImage**, il se peut que toutes les informations ne soient pas afficher sur l'interface.

Au niveau des annotations, la classe **Team** permet de différencier tous les robots en fonction de leur équipe et permet deux robots avec le même numéro mais aussi un nombre illimité d'équipe.

Il sera bien évidemment conseillé dans ce cas là de rajouter des couleurs dans le fichier *annotation\_settings.json*.

Pour l'interface, l'affichage des différentes informations sur le côté est équipé d'un scroll, qui nous permet de nous déplacer dans l'équipe comme on peut le voir ici en simulant des robots.



FIGURE 3.8 – Interface avec scroll

# Chapitre 4

## Fonctionnalité implémentées

### 4.1 Les annotations

Nous avons implémenté différentes annotations qui sont incrustables sur des vidéos. Si on lance l'exécutable de **annotateImage**, les choix d'annotations se font seulement avant le lancement du programme.

Ces choix se font dans le fichier *annotation\_settings.json*. Si on lance l'interface, les choix d'affichage s'initialisent grâce au fichier Json mais peuvent être modifiés pendant la lecture de la vidéo. Nous allons suivre l'architecture du fichier Json pour expliquer les différentes fonctionnalités implémentées pour les annotations.

Le booléen "write" présent pour chaque annotation exprime l'écriture ou non sur l'image de l'annotation.

Il faut également récupérer les numéros des équipes. Soit ils sont connus, soit il faut les récupérer avec l'exécutable *init\_match* du package `annotateImage` explique 4.2.1, page 25.

#### 4.1.1 Les couleurs

Les deux premiers éléments du fichier json sont la couleurs de chaque équipe. Par défaut, les couleurs des équipes sont bleu pour l'équipe 1 et rose pour l'équipe 2, ce qui correspond aux couleurs des robots en match.

### 4.1.2 La position

La position permet d'afficher un cercle montrant la position où le robot pense être. Elle s'affiche forcément pour tous les robots à la fois ou aucun. Il est possible de choisir le rayon du cercle (en pixel) et l'affichage ou non du numéro du robot au centre de la position.

La taille de l'écriture du numéro est proportionnelle à  $\frac{3}{4}$  du rayon du cercle. Il est donc déconseillé de l'afficher si la taille du cercle est minime.

La position s'affiche simplement en transformant le point du plan réel en point sur l'image grâce à la fonction `fieldToImg` fournie par le client.

### 4.1.3 La direction

L'annotation de la direction dessine une flèche montrant la direction vers laquelle le robot avance. La flèche démarre de la position du robot. Il est possible de changer sa taille.

La flèche est créée en définissant un second point sur le dessin dans la direction donnée (en radian). Ce point est déterminé en additionnant les coordonnées de la position à  $\cos(\text{direction})$  pour l'axe des  $x$  et  $\sin(\text{direction})$  pour l'axe des  $y$ .

Ensuite, nous transposons nos deux points dans le plan de l'image grâce à `fieldToImg`.

Enfin, nous calculons la distance entre ce nouveau point et la position grâce au théorème de Pythagore, puis nous réduisons proportionnellement cette distance à la taille de la flèche souhaitée.

Nous affichons la flèche de la couleur de l'équipe du robot. Il se peut que la flèche s'affiche en noir, cela signifie que l'angle donné par le robot est supérieur  $2 * \pi$ . Ceci témoigne d'une imprécision du robot. Nous pouvons voir cette erreur dans la vidéo 2vs1.



FIGURE 4.1 – Erreur de direction d'un robot

#### 4.1.4 La trace

L'annotation de la trace est représentée par de multiples cercles modélisant les anciennes positions du robot. La trace ne peut être visible que sur un robot à la fois, il peut être changé en cours de match lorsque la visualisation est lancée avec l'interface.

Elle s'affiche de la couleur de l'équipe du Robot choisi mais en plus foncé comme on peut le voir dans la Figure 4.2. Nous pouvons choisir la taille de la trace, il est préférable de l'avoir plus petite que la position.

La trace nous montre les dernières positions depuis un certain nombre de secondes défini dans *annotation\_settings* grâce à "delay\_old\_pos". La trace est stockée dans une map.

Pour afficher la trace nous affichons donc toutes les positions présentes dans la map pour un certain nombre de secondes précédentes.



FIGURE 4.2 – Affichage de la trace

#### 4.1.5 La balle

Tout comme la trace, nous ne pouvons voir la balle du point de vue que d'un seul robot à la fois. Nous pouvons choisir la taille de la balle. La couleur de la balle est définie arbitrairement en gris.

Ce qui a été compliqué avec la balle c'est que la position de la balle est donnée dans le repère du robot, non du terrain.

En effet, dans les logs (Figure 3.1), la position du robot est définie grâce *self\_in\_field* donc dans le repère du terrain mais la position de la balle est dans *ball\_in\_self* donc d'après le repère du robot.

Nous avons donc du changer de repère grâce à une équation de plan.

Soit  $(x_B, y_B)$  les coordonnées de la balle dans le repère du robot et  $(x'_B, y'_B)$  dans le repère du terrain. On définit également  $(x_R, y_R)$  les coordonnées du robot dans le plan du terrain et  $\theta$  l'angle de la direction du robot en radian.

L'équation est donc :

$$\begin{cases} x'_B = x_R + x_B \cos(\theta) - y_B \sin(\theta) \\ y'_B = x_R + x_B \sin(\theta) + y_B \cos(\theta) \end{cases}$$

#### 4.1.6 La position souhaitée (Target)

La target représente la position que le robot souhaite atteindre. Elle ne s'affiche donc que pour un robot à la fois.

Elle est marquée par une croix dont la taille est définie par l'utilisateur dans "target\_size". Si on a également la position du robot, une ligne en pointillé s'affiche entre la position du robot et celle souhaitée. Il est possible de ne pas afficher cette ligne en mettant un nombre plus grand que la taille de l'image (avec les vidéos données, 1000 convient parfaitement).

La taille des traits est égale à la taille des espaces, pour tracer la ligne, nous avons fait un itérateur qui colorise un certain nombre de pixel pour le trait puis qui laisse l'espace de même taille.



FIGURE 4.3 – Affichage de la target

#### 4.1.7 Les lignes du terrain et le score

Nous pouvons choisir d'afficher les lignes du terrain en fournissant le fichier Json définissant les caractéristiques du terrain sur lequel le match est joué. La fonction d'affichage des lignes a été fournie par le client. Ce fichier Json est nécessaire pour la calibration de la caméra.

Comme nous en avons vu Figure 3.2 à la page 13, les lignes du terrain peuvent parfois avoir des erreurs à cause de la distorsion de la caméra.

Afficher ces lignes peut nous aider à comprendre les différences entre la position annotée et réelle du robot.

Il y a une possibilité d'afficher le score dans un des coins de l'écran. Les valeurs à donner en x et y sont données en commentaire dans le Json. Nous n'avons pas défini de coin par défaut car il peut changer à chaque vidéo. Par exemple, l'idéal pour la vidéo static de 2vs1 est dans le coin en haut à gauche mais si on l'affiche là, le score ne se voit pas sur la vidéo webcam du 21/03/2019.

Il faut noter que le score s'affiche automatiquement dans l'interface.

#### 4.1.8 Délai et optimisation des annotations

Les deux derniers éléments du fichier Json sont "delay\_annotation" et "optimized". Le premier exprime le temps en seconde durant lequel un message est considéré comme valide. C'est à dire le temps pour lequel on va l'afficher avant qu'il soit trop vieux.

L'optimisation permet d'afficher les annotations avec une opacité reflétant la durée. Un message plus vieux sera plus transparent qu'un message tout juste reçu.

Pour l'ajout de certaines annotations comme la trace, nous avons souhaité utiliser la transparence. Ainsi, les anciennes positions deviennent graduellement transparentes, jusqu'à disparaître.

Nous avons donc décidé d'utiliser la fonction d'OpenCV (**AddWeighted**) permettant de fusionner deux images en fonction d'une opacité.

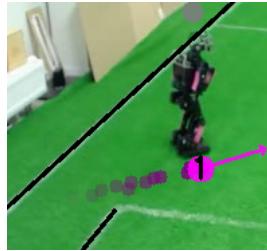


FIGURE 4.4 – Affichage de la trace optimisé

La transparence des objets se fait en trois étapes.

Tout d'abord, on calcule l'opacité en fonction du temps.

Ensuite on copie l'image dans une image provisoire (overlay) et on ajoute l'annotation à cette copie. Ensuite on ajoute l'image copiée et annotée à l'image de départ (blend) en fonction de l'opacité calculée grâce à la fonction OpenCV.

La différence de performance entre les annotations non optimisées et optimisées est calculée dans les tests sur la performance en 5.3, page 32.

## 4.2 Les outils de `annotateImage`

En dehors des tests, nous avons deux fichiers exécutables dans la partie `annotateImage`.

### 4.2.1 `Init_match`

Tous les choix sur les numéros de robot et d'équipe peuvent être difficiles si on ne connaît pas les robots de la vidéo. L'exécutable `init_match` permet de lire les 30 premières secondes de vidéo et de connaître donc les robots présent en jeu ainsi que leur équipe. On peut en voir le résultat Figure 4.5 :

```
In this game :
Robot n° 2 , team n° 9
Robot n° 4 , team n° 9
Robot n° 1 , team n° 12
```

FIGURE 4.5 – Résultat de l'exécutable `init_match`

Tout comme les tests, il faut activer les tools de `annotateImage` dans le CMakeList pour avoir l'exécutable `init_match`.

### 4.2.2 Main\_annotationImage

Notre fichier principal de lancement est l'outil *main\_annotationImage*.

Il permet de lancer un monitoring et d'afficher une vidéo simplement, comme cela a été détaillé dans l'architecture du projet.

Il y a une option ajustable pour le fichier *main\_annotationImage* dans le fichier de configuration *match\_settings*. Il s'agit de l'élément "speed\_optimized". Si, l'utilisateur met ce booléen à false, alors le match défilerà le plus vite possible pour l'ordinateur. Sinon, le match défilerà à vitesse normale, c'est à dire à la même vitesse que le fichier vidéo fourni.

Pour cela, nous calculons le temps réel écoulé depuis le lancement de la vidéo grâce à deux chronos. Nous comparons ensuite ce temps écoulé au temps écoulé de la vidéo.

Si nous avons de l'avance par rapport à la vidéo alors nous ralentissons grâce à un temps d'attente avant la prochaine image.

Sinon nous passons immédiatement à l'image suivante. En cas de retard trop important, nous sautons une image ce qui nous permet de ne pas avoir de retard par rapport à la vidéo.

## 4.3 L'interface

Nous ajoutons des annotations sur une vidéo de match. Cependant, étant donné le grand nombre d'informations, l'image peut se trouver saturée et devenir illisible. Notre interface doit résoudre ce problème en proposant des options d'affichage qui conviennent à différents usages.

### 4.3.1 Choix de l'API

Dans un premier temps, il a fallu choisir une API pour créer l'interface de notre programme. Nous en cherchions une plutôt facile à prendre en main, en C++, et qui propose suffisamment d'options pour créer une interface complète.

Nous avons d'abord pensé à **OpenCV**, que nous utilisons déjà pour les annotations. La prise en main est assez rapide mais les options sont trop peu nombreuses. Les interfaces en **OpenCV** se limitent principalement à afficher des images, sans aspect esthétique.

**GTK** et **FLTK** ont été rapidement considérés, mais la prise en main semblait trop compliquée et aurait pris trop de temps.

Nous avons donc choisi **QT** dans sa version 5.9, qui est très bien documenté [ref, 2012], facile à prendre en main et présente un très large éventail d'options pour créer une interface riche et complète.

Au début du projet, nous avons rencontré des difficultés pour faire communiquer l'interface et les fonctions d'annotation. Après la refonte de l'architecture du projet, la liaison entre les modules interfaces et **annotateImage** était grandement simplifiée, et nous avons pu lire une vidéo dans notre fenêtre **Qt**.

Nous avons ensuite pu ajouter des options pour choisir les annotations à afficher, modifiables pendant la lecture de la vidéo.

De chaque côté de la vidéo, nous affichons des informations sur le match, les robots, les équipes et les choix d'annotations.

### 4.3.2 Affichage de la vidéo

On ne peut pas afficher d'image cv::Mat dans un label d'une interface QT. Ce format d'image est toutefois celui nécessaire pour les annotations. Nous convertissons donc l'image produite par nos fonctions de traitement en QImage, puis en QPixmap. Elle est ensuite placée dans le label idoine.

Pour rafraîchir l'image, un QTimer fait appel à la fonction **changeImage** avec un intervalle de *SPD\_INTERVAL* millisecondes. Cette fonction reprend le fonctionnement de *main\_annotationImage*, mais au lieu d'afficher la vidéo de sortie, les images sont placées dans **labelVideo**.

### 4.3.3 Les fonctionnalités de l'interface

Lorsque nous lançons l'interface, la configuration des annotations par défaut sera lu dans le fichier *annotation\_settings.json*.

Nous voyons alors s'afficher la fenêtre d'interface comme ci-dessous.



FIGURE 4.6 – Fenêtre d’interface

Les deux panneaux à droite et à gauche de la vidéo sont créés grâce aux classes **TeamPanel** et **RobotPanel**. Ils contiennent les informations sur les annotations en cours.

En dessous de la vidéo, nous avons le slider expliqué un peu plus bas, partie 4.3.4.

Ensuite, nous avons les 3 boutons de l’interface.

Le premier est un simple bouton play/pause qui permet de faire des arrêts sur image. (Très pratique par exemple pour prendre des situations en photos, voir Figure 4.1)

Le second bouton est un bouton FastForward permettant de changer la vitesse d’affichage de la vidéo.

Au démarrage, la vidéo est en 30ms/img, une vitesse normale. En activant le bouton  $\gg$ , la vitesse de la vidéo passe à 1ms/img, c’est-à-dire que la vitesse de la vidéo est régulée par la vitesse des annotations.

C’est l’équivalent du booléen *speed\_optimized* de *match\_settings* nous permettant de faire varier la vitesse de la vidéo dans le fichier *main\_annotationImage*.

Enfin, le dernier bouton permet de faire apparaître une fenêtre pop-up permettant de changer les annotations (Figure 4.7).

Nous voyons que les robots affichés pour les annotations sont les mêmes que ceux de la vidéo.

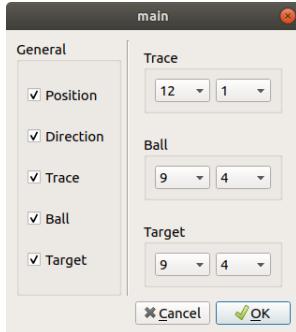


FIGURE 4.7 – Fenêtre de choix d’annotations

#### 4.3.4 Le slider

Un slider a été implémenté pour choisir un moment précis de la vidéo. Il prend des valeurs allant de 0 à 99.

Pour que le slider avance automatiquement lorsque la vidéo est en lecture, on calcule le pourcentage d'avancement à partir du *time\_stamp* de l'image actuelle et de la durée totale de la vidéo.

Lorsque le slider est déplacé, le *time\_stamp* correspondant à l'image et aux messages à lire est incrémenté (respectivement décrémenté) jusqu'à atteindre la position indiquée par le slider. Seul un entier est modifié dans une boucle while, nous ne parcourons pas toutes les images et messages entre les deux *time\_stamp*. Pour l'utilisateur, la latence est donc négligeable.

La position maximale du slider est donc 99, ce qui correspond donc à 99% de la vidéo. Il reste donc encore 1% de à lire à ce moment.

On affiche le temps écoulé dans un label en dessous du slider. Ce temps est exprimé en minutes et secondes. Les matchs ne devraient pas durer une heure, notamment à cause de la batterie des robots.

Il nous a semblé plus pertinent de simplifier l'affichage en ayant seulement deux nombres (minutes et secondes) plutôt qu'un nombre pour les heures qui serait égal à 0 dans la majorité des cas. Si le match dure plus de 60 minutes, le nombre de minute continuera d'incrémenter.

Arrivé à la fin de la vidéo, ce n'est plus le temps qui s'affiche mais "end of video".

# Chapitre 5

## Tests effectués

Nous n'avons pas effectué de test sur les fonctions fournies par le client et les fonctions **OpenCV** que nous avons utilisé pour afficher les annotations.

Il est pour nous compliqué de faire des tests sur l'exactitude de notre affichage. Nous sommes donc concentrés sur des tests fonctionnels.

### 5.1 Tests sur l'utilisation du Json

Tous les réglages des annotations s'effectuent dans le fichier *annotation\_settings.json*. Nous avons donc testé les différents problèmes que nous pouvons rencontrer dans la lecture de ce fichier Json.

**Il manque un item dans le fichier** : Pour tous les éléments utilisés nous utilisons une fonction *CheckMember* qui va vérifier qu'il ne manque pas d'informations dans le fichier Json.

Il faut noter que même si une annotation n'est pas utilisée, nous vérifions que tous ses items sont là. En effet, grâce à l'interface nous pouvons changer les annotations sans relancer notre exécutable. Il est donc important de vérifier que tous les objets du Json soient présent dès le début afin de pouvoir les utiliser si besoin en changeant les annotations.

**Un item a été ajouté en trop** : Les éléments présents dans le fichier *annotation\_settings.json* mais non utilisés dans nos fonctions ne gênent en aucun cas l'ajout des annotations dans la vidéo. Ils sont justes ignorés.

**Un item est présent deux fois dans le fichier** : Nous avons testé ce cas là en dupliquant des éléments du Json. Dans ce cas-là la dernière itération de l'élément sera celle prise en compte.

## 5.2 Tests sur la lecture des logs

Le problème principal sur les logs que l'on peut trouver en match est l'interruption de l'envoi des messages par les robots. Les tests sur l'interruption des logs s'effectuent dans le fichier *test.cpp*, dont l'exécutable est *tools/test\_logs*.

Pour cela nous avons eu besoin d'identifier les dépendances entre les différentes annotations.

- **La Position** : rien ;
- **La Direction** : Nécessite la Position ;
- **La Balle** : Nécessite la Direction et la Position ;
- **La Trace** : Nécessite au moins une Position ;
- **La Position souhaitée** : Nécessite la Position actuelle pour avoir le trajet à effectuer par le robot, sinon rien.

Si nous nous plaçons maintenant dans la fonction **addAnnotation** de la classe **Annotation** nous pouvons voir l'application de la hiérarchie dans les annotations.

La trace est annotée en premier elle se place donc à l'arrière-plan, toutes les autres annotations seront ajoutées par dessus. C'est également la seule qui peut être affichée même si l'il n'y a pas de messages (si on choisit un temps d'expiration du message inférieur au temps désiré pour la trace).

Ensuite, nous vérifions si le dernier message du robot est encore valide puis nous ajoutons les annotations en lisant dans le message et en organisant le plan (par exemple, la position est affichée après la direction pour que la flèche n'efface pas le chiffre).

Avant d'afficher une position, une fonction **isPosValid** vérifie en fonction des paramètres de la caméra que la position n'est pas en dehors du cadre.

En cas de positions incorrecte, nous n'allons donc pas l'afficher.

Nous vérifions les positions après le passage du plan réel au plan de l'image (après avoir appelé la fonction **fieldToImg**).

Nous avons trouvé dans la vidéo du 21/03/2019 des positions absurdes. La position de la target du robot numéro 1 de l'équipe 3 (en rose) alterne régulièrement entre les points  $(-2.1685e+12, 2.02621e+12)$  et  $(-27729.4, -74940.3)$ , tous les deux totalement hors champs

Enfin, dans le fichier de test nous interrompons les différentes annotations à des temps différents pour chaque message. La reprise des annotations après interruption ne présente pas de problème sur notre fichier de test.

Il y a cependant quelques faits intéressants à noter sur les logs des robots.

Si on dé-commente les lignes 82 et 99 du fichier *test.cpp*, cela nous affiche les *time\_stamp* du **GC\_Msg** et des **Robot\_Msg**. On voit donc à ce moment que chaque message s'actualise à des temps différents et que les messages sont donc utilisés pour plusieurs frame de la vidéo. C'est pour cela que nous avons ajouté une ligne permettant d'optimiser la performance en empêchant la réécriture d'un même message.

Il est important de noter que le **Robot\_Msg** se décompose en deux parties. Pour accéder au **Robot\_Identifier**, il faut appeler **robot\_entry.first** et le reste du message (**Robot\_Msg**) est accessible depuis **robot\_entry.second**. (voir Figure 3.1, page 12)

## 5.3 Tests sur la performance

### 5.3.1 Fichiers de test

Les tests sur la performance de notre programme s'effectuent dans le fichier *test\_time.cpp* et *test\_video.cpp*.

Ce test crée un fichier *résultat.csv* comprenant le temps des 5 éléments majeurs du programme : Récupération des messages, récupération de l'image, clonage de l'image, ajout des annotations lignes du terrain et score, ajout des autres annotations.

Ce programme va donner les résultats sur une vidéo. Si le fichier *replay.json* en contient une seule ce sera celle-ci, s'il en contient 2, ce sera la seconde dans l'ordre alphabétique.

Ceci est important car nous avons effectué des tests avec les vidéos de 2vs1, pour lesquels il est plus intéressant d'avoir les informations de *static\_video* plutôt que *dynamic\_video* (qui n'affiche pas les annotations).

Le fichier de test, va lire les 1000 premières images de la vidéo ce qui nous permet d'avoir un résultat rapide. Ce chiffre peut être changé au début du fichier *test\_time.cpp*, c'est la variable **NB\_FRAME\_READ**.

Nous avons un dernier fichier de test appelé *test\_video.cpp* qui est un script permettant de lancer 5 fois le programme *main\_annotationImage* et le fichier de test, *test\_time*. Elle peut nous permettre de tester la performance en lançant de multiples fois le programme.

### 5.3.2 Paramètres des tests

Pour mesurer les performances, nous avons utilisé 4 machines de performances inégales afin d'être le plus exhaustifs possible quand aux performances que l'utilisateur peut espérer.

La vitesse d'exécution du programme dépend principalement du processeur, de la vitesse du disque dur (SSD préférable), et de la carte graphique. La performance des composants influe grandement sur le fonctionnement de notre programme. Afin de savoir où situer une machine par rapport aux résultats, nous listons ici les scores **GeekBench 4** des machines de test.

TABLE 5.1 – Scores GeekBench 4

Ordinateur :	A	B	C	D
single core :	2500	4200	5000	5500
multi core :	4000	12500	15000	16000

Pour les tests suivants, nous traiterons les 1000 premières images de vidéos de matchs (résolution 640\*480 pixels).

### 5.3.3 Test de vitesse d'annotation

Avant de commencer ce projet, nous avions réalisé des tests pour déterminer les performances que nous pouvions espérer. Lors de ces tests, il était ressorti que l'utilisation des fonction de dessin d'**OpenCV** était très peu coûteuse en temps de calcul.

La majorité du temps de traitement était le chargement (ou création) et l'enregistrement de l'image. Nos prévisions à l'époque étaient des capacité de traitement de l'ordre de 80 mégapixels par secondes dans le pire des cas.

### 5.3.4 Performance selon le codec vidéo

Les fichiers vidéos fournis par le client sont plutôt volumineux (environ 3 mb/s pour une résolution de 640\*480 pixels). Les vidéos sont stockées dans des conteneurs .avi, nous avons donc supposé qu'il était possible d'en réduire la taille en utilisant des codecs plus récents. Nous avons converti les vidéos dans les formats MP4, encapsulé dans des conteneurs .mp4 et .mov.

Le gain de poids a été assez conséquent, avec une réduction d'un facteur environ 10 sans trop dégrader l'image.

Les données des fichiers issus de ces conversions sont davantage compressées que celles de départ, il nous a semblé pertinent d'évaluer l'impact de ce nouveau codec sur les performances.

Le test est effectué avec *test\_video*, qui nous donne la durée pour récupérer, annoter et afficher une image. La vidéo de test est celle sans mouvement de caméra de 2vs1.

Pour ce test, nous lisons trois fois deux vidéo (caméra en mouvement et immobile) simultanément. Nous mesurons le temps de traitement de chaque image d'une vidéo (en microsecondes). Nous enregistrons également le nombre de framedrop.

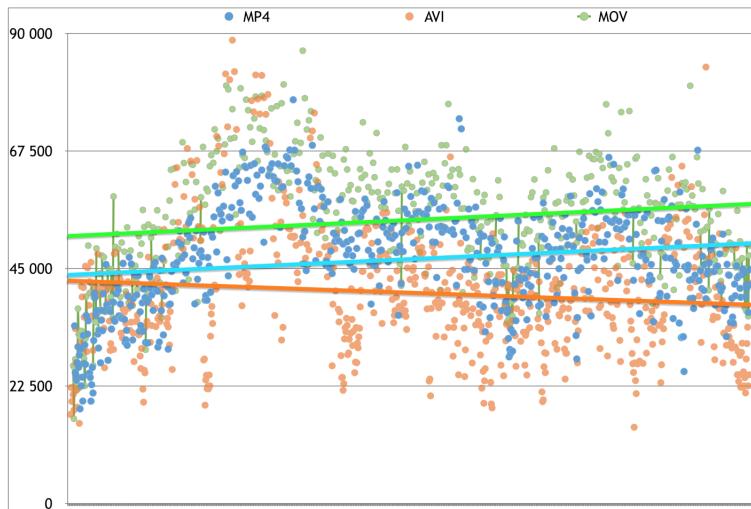


FIGURE 5.1 – Performances en fonction du codec

TABLE 5.2 – Framedrop en fonction du format

avi	mp4	mov
33%	44%	52%

On observe que les fichiers les moins compressés (avi) permettent de meilleures performances. Le poids de la vidéo reste important mais cela nous semble un sacrifice acceptable. L'impact sur les performances des autres formats est non négligeable, et peut être handicapant sur une petite configuration.

Pour la suite de ces test, nous utiliserons les vidéos avi des clients.

### 5.3.5 Impact des blend

Pour certaines annotation, nous effectuons des "blend" d'images. C'est le cas par exemple pour la trace. Cette opération est coûteuse en ressources (bien plus que de dessiner les annotations).

Ici, nous comparons les performances des quatre ordinateurs pour le traitement d'une vidéo, avec et sans l'utilisation de blend. Nous lisons simultanément la vidéo avec mouvement de caméra pour simuler les performances d'une interface avec deux flux vidéo dont un annoté.

Le test ici est effectué avec l'exucatable *test\_time*.

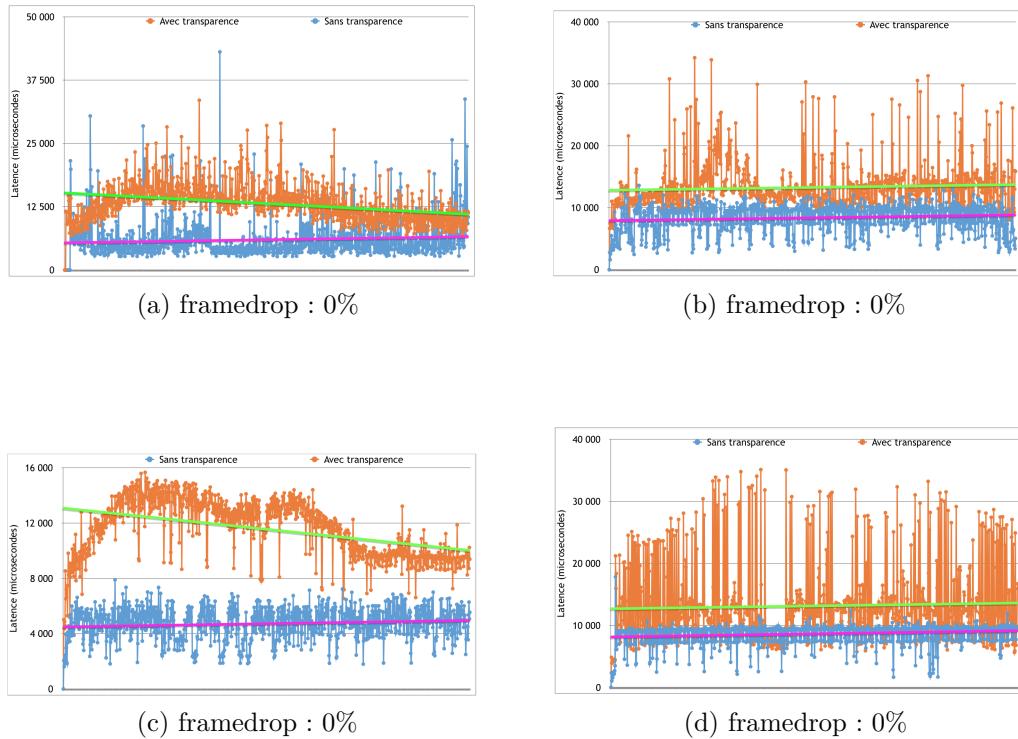


FIGURE 5.2 – Impact du blend sur les performances, deux vidéos

Nous travaillons ici avec des vidéos 30fps, il faut donc que le temps de traitement reste inférieure à 33 ms pour éviter les sauts d'images (ou ralentissements). On observe que tous les ordinateurs parviennent à rester en dessous de cette limite. En moyenne, sur les 4 machines, pour deux vidéos simultanées, l'utilisation de blend augmente le temps de traitement de 80%.

Mis à part de rares pics, nous n'observons pas ici de framedrop. Pour avoir une idée des limites de notre programme dépendamment de la configuration de la machine, d'autres tests ont été menés.

### 5.3.6 Limites de puissance

Ici, nous donnons aux ordinateurs 6 vidéos avec puis sans blend à traiter simultanément, fichier *test\_video*. Le but est de déterminer les limites du programme.

Sur ces graphiques, deux points reliés sont des images qui se suivent. Dans le cas d'un framedrop, la ligne n'est pas présente.

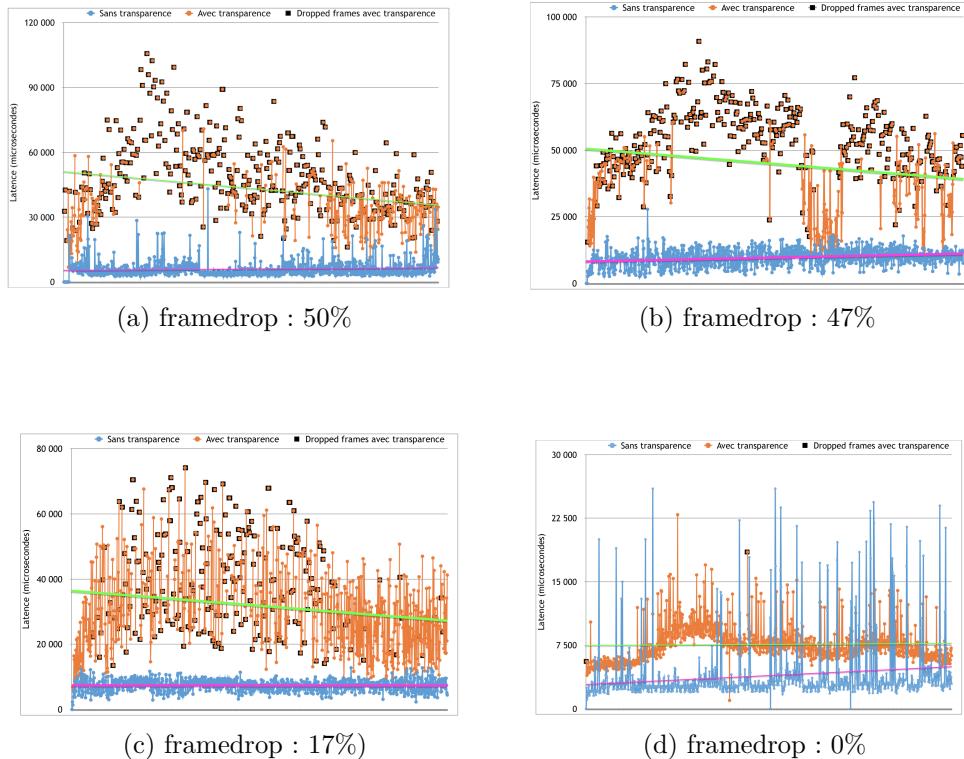


FIGURE 5.3 – Limites de puissance

Ici nous voyons que les ordinateurs A et B ne parviennent pas à proposer un traitement en temps réel avec blend sur 6 vidéos à 30 images par secondes de manière fluide. Avec 50% de frame drop, la vidéo tombe à environ 15 fps et n'est plus très agréable à regarder. Les performances de l'ordinateur C sont encore convenables. L'ordinateur D, contenant les composants les plus récents, ne semble pas affecté par l'augmentation du nombre de vidéos.

Pour le traitement des 6 vidéos sans blend, aucun framedrop n'a eu lieu (à part pics sporadiques). Les valeurs de latence sont en moyenne de 4.4ms, très éloignées des 33ms, quand la coupure s'effectue.

Nous avons continué les expérimentations sur l'ordinateur D pour déterminer le nombre limite de vidéo qu'il pouvait lire de manière fluide.

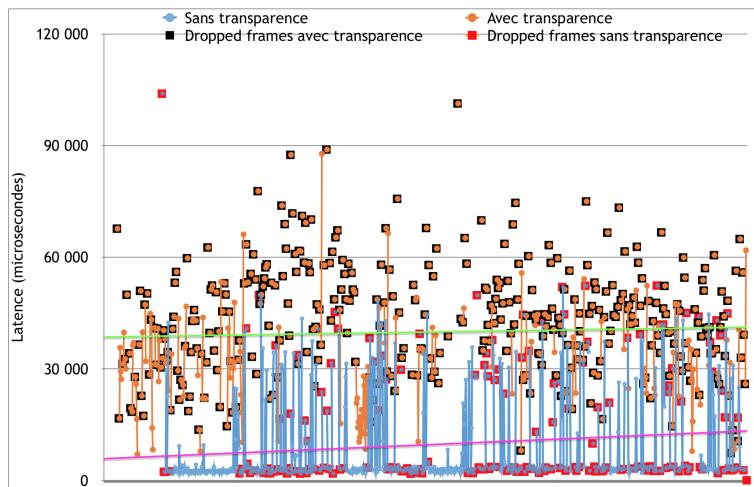


FIGURE 5.4 – Limites de performances ordinateur D

Dans le graphique ci-dessus, nous avons mesuré les performances de traitement avec blend (transparence) pour 20 vidéos. à partir de ce nombre, le framedrop devient supérieur à 50%.

Pour le traitement sans blend, nous avons mesuré sur le même graphique les performances pour 40 vidéos simultanées. Le framedrop était de 40% environ.

L'ordinateur D est celui avec les composants les plus récents (2017). Ceci semble le favoriser grandement par rapport à l'ordinateur C, de puissance presque similaire. Si le traitement ne tourne pas de manière fluide, il faut donc considérer l'achat de nouveaux composants.

En se basant sur ces résultats et ces proportions de framedrop, voici performances que l'on peut espérer des ordinateurs avec le blend :

TABLE 5.3 – Puissance de traitement des ordinateurs (mégapixels/s)

A	B	C	D
20	35	50	80

Lors de la rédaction du cahier des besoins, les tests de l'époque nous faisaient estimer une performance d'environ 80 mégapixels par seconde sur l'ordinateur C. Ces estimations étaient pessimistes, et se basaient sur l'incrustation d'une quantité absurde d'annotation. Le blend toutefois est plus coûteux que le dessin sur l'image. Ainsi, l'ordinateur C atteint 50 mégapixels/s avec blend. On a observé un gain de performance d'environ 80% sans blend. Ces résultats sont donc cohérents avec nos prédictions.

# Chapitre 6

## Problèmes rencontrés et limites

### 6.1 Les vidéos à annoter

#### 6.1.1 Le live

Nous avons testé notre programme seulement sur des vidéos en replay, nous ne savons pas comment réagi le problème à un flux vidéo en direct.

Nous savons en revanche que l'outil de base de visionnage fourni par le client *basic\_monitoring* fonctionne en live. Nous pensons que notre programme peut fonctionner en live sans problème mais peut-être avec du délai dans les annotations (surtout si l'utilisateur veut annoter plusieurs vidéos en même temps et ajouter la transparence).

#### 6.1.2 Taille des vidéos

Une limite imposée par le code du client est la taille des vidéos. Elles doivent impérativement être en format 640\*480.

Il est possible de lire des vidéos en dehors de ce format si on désactive l'affichage des lignes du terrain, mais dans ce cas les annotations affichées seront inexactes.

Comme nous pouvons le voir dans les deux images ci-dessous (Figure 6.1), le décalage est nettement visible pour la vidéo b, trop large par rapport au format demandé. La position du robot 4 et sa target sont décalés par rapport au robot et aux lignes du terrain

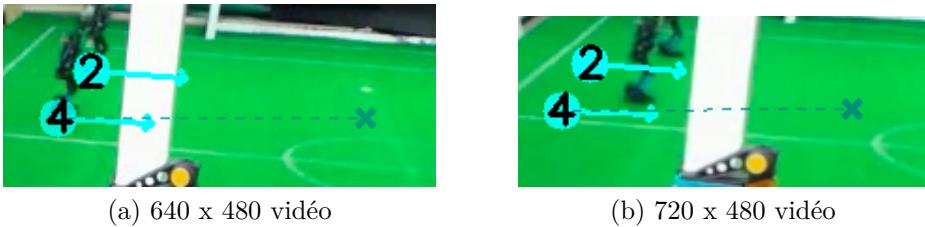


FIGURE 6.1 – Différence d’annotation en fonction de la taille de la vidéo

### 6.1.3 Format des vidéos

Nous avons pu tester différents formats de vidéo (avi, mov, mp4). Ils sont tous les trois acceptés par OpenCV. Nous avons observé des différences de performances suivant le format, nous avons donc effectué des tests.

Le test est expliqué en partie 5.3, page 32).

Ces tests confirment bien une différence de performance des codecs. Si la charge de traitement est importante, nous conseillons donc d’utiliser le format avi comme proposé dans les dossiers de vidéo du drive.

Enfin, nous ne prenons pas en charge les vidéos dynamiques, elles ne seront pas annotées. Pour la vidéo dynamique, la condition **isFullySpecified()** n’est pas remplie. La vidéo étant en perpétuel mouvement, nous ne pouvons pas (dans l’état actuel du programme) recalculer d’équation de plan.

Pour les vidéo grand angle (par exemple ptgrey.avi dans les vidéos du 20/02/2019 et du 21/03/2019) des annotations parfois inexactes (voir Figure 3.2, page 13).

## 6.2 La transparence

Nous souhaitions utiliser des annotations plus ou moins transparentes (principalement pour la trace mais également pour d’autres).

Malheureusement, les fonctions de dessin d’OpenCV ne gèrent pas la transparence. OpenCV gère bien les images avec transparence, les fonctions de dessin acceptent des couleurs avec transparence, mais lors du dessin, elles ignorent cette valeur, ne considèrent que les canaux R,G,B et dessinent donc en opacité maximale.

La solution trouvée (qui semble être la seule proposée par OpenCV) est le blend d’images entre elles (voir 4.1.8).

## 6.3 Le slider

Pour implémenter un slider qui permette de choisir un moment dans la vidéo, il est nécessaire de connaître certaines caractéristiques de celle-ci : Soit le nombre total d'images et le nombre d'images par seconde, soit la durée totale de la vidéo et le nombre d'images par seconde.

Dans un premier temps nous avons voulu utiliser le nombre total d'images, mais le framerate des vidéos encodées par les clients (30fps dans la plupart des cas) n'est pas constant, avec parfois moins de 30 images sur une seconde. Dans ce cas, certaines images sont dupliquées dans le fichier vidéo pour maintenir la fréquence.

Mais ces images "doublons" ne sont pas comptabilisées dans le nombre total d'image. On ne pouvait donc pas précisément définir la durée de la vidéo.

La conséquence était un slider qui finissait sa course alors qu'il restait une portion non négligeable de la vidéo à lire (jusqu'à 50% dans nos vidéos de test).

Le nombre d'images dupliquées dépend de la vidéo, et est dû (pensons-nous) à un manque de puissance au niveau de l'ordinateur l'ayant encodé lors de la capture.

C'est pourquoi notre slider a donc été implémenté grâce aux `time_stamp` des images.

## 6.4 La trace

Il a été très difficile de stocker les anciennes positions de manière optimisée.

Nous avons au début créé une queue. Pour afficher la trace on affichait la première position puis on l'insérait à la fin jusqu'à avoir parcouru toutes les positions et revenir à la position initiale.

Cette méthode n'était plus adaptée une fois le slider implémenté. En effet, la queue ne permettait pas du tout le retour en arrière. Si l'utilisateur effectuait un retour en arrière, il était impossible d'afficher correctement la trace et il fallait recommencer son affichage à partir de la position actuelle.

Nous avons donc implémenté une map où l'on stocke les positions en fonction de leur `time_stamp`. Cette méthode a fonctionné parfaitement avec le slider.

Lorsque nous parcourons la vidéo, nous stockons la position dans la map et nous affichons toutes les positions dont le *time\_stamp* est compris entre le temps actuel et le nombre de seconde précédent en fonction du nombre de *delay\_old\_pos* choisi par l'utilisateur.

Nous ne supprimons donc jamais les anciennes positions des robots afin d'avoir un affichage optimisé dans l'interface. Mais cette mémoire importante n'affecte pas à priori les performances, pour les vidéos sur lesquelles nous l'avons testé.

# Chapitre 7

## Conclusion

Grâce à nos deux modes d'affichage et nos multiples annotations, notre outil de visionnage permet de mieux saisir l'état du jeu tel qu'il est perçu par un robot.

On peut imaginer une interface dédiée aux développeurs qui serait centrée sur un robot ou une équipe. Cela leur permettrait d'avoir des informations précises plus détaillées et complètes qu'actuellement.

Notre interface ne traite qu'une seule vidéo à la fois. Pouvoir changer de point de vue lors d'un match serait un atout pour pouvoir mieux suivre les actions.

Nous avons préféré rester sur une interface plutôt générale mais bien fonctionnelle (scroll dans les équipes, pop-up pour le choix des annotations et slider). Ajouter les annotations à l'interface au fur et à mesure a été moins compliqué que d'ajouter ces outils ergonomiques.

Pour ce qui est des annotations, nous avons longtemps parlé de faire des ellipses à la place des cercles.

L'idée était exprimer la dispersion des probabilités de positons qu'un robot peut envoyer. Ces ellipses pouvaient aussi représenter la distorsion de la caméra, nous permettant de mieux comprendre l'angle de vision.

Finalement, nous n'avons jamais implémenté l'ellipse, n'ayant de la part du robot qu'une simple position ( $x,y$ ). Nous n'avons pas jugé l'implémentation de l'ellipse prioritaire, nous avons préféré ajouter des annotations.

Nous avons pensé à ajouter les informations sur le jeu dans l'interface (temps restant en touche pour robot exclu, nombre de cartons jaune et rouge par robot). Ces informations étaient d'une utilité limitée dans les premières vidéos tournées (pas de buts). Nous avons trouvé plus pertinent d'afficher à la place les choix des annotations.

En revanche, si l'on stocke les **GCRobotMsg**, il paraît plutôt simple d'afficher ces informations. Il suffit de lire le message dans la classe **RobotInformation**, puis lire le message grâce aux fonctions proto déjà implémentées.

Ajouter des annotations comme les penalty, la position des adversaires ou encore la cible de la frappe est assez rapide. Ce sont des simples cercles à afficher comme la balle ou la position du robot.

Enfin, il aurait été intéressant de pouvoir enregistrer une vidéo annotée. Nous avions testé l'enregistrement d'une vidéo à partir d'images lors du premier rendu du cahier des besoins.

Nous n'avons pas adapté notre code de test aux fichiers du client, mais nous avions réussi à enregistrer à partir d'une liste d'image.

Au terme de ce travail, nous avons pu créer un outil de visualisation de match de football robotique qui répond à une majorité des besoins du client. Notre projet est destiné à évoluer grâce aux nombreuses extensions possibles.

# Bibliographie

- [ref, 2012] (2012). Qt documentation 5.9. <https://doc.qt.io/qt-5.9/index.html>.
- [Gary Bradski, 2008] Gary Bradski, A. K. (2008). *Learning OpenCV : computer vision with the OpenCV Library*. O'Reilly Media.
- [Kannala J., 2006] Kannala J., B. S. (2006). A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.28(8), pages 1335–1340.
- [Matas J., 2000] Matas J., Galambos C., K. J. (2000). Robust detection of lines using the progressive probabilistic hough transform. *Computer Vision and Image Understanding*, 78(1), page 119–137.

## Annexe A

# Architecture des messages

Légende :

- Chaque carré bleu correspond à ce qui est contenu dans le fichier proto dont le nom est marqué dedans.
- Il y a forcément au minimum un **GC\_Msg** ou **Robot\_Message**, c'est pourquoi il sont en orange.
- Les éléments en rouge sont "required", en effet, si on prend le début du message, il est normal que chaque message ait un identifiant.
- Les fichiers en vert sont "repeated", en effet, on a plusieurs Message. Dans chaque message on va avoir plusieurs équipes et robots (donc **GCTeamMsg** et **GCRobotMsg** sont en vert).
- Enfin, les éléments en jaune sont optionnels.
- La position définie dans le fichier *position.proto* est réutilisée dans tous les éléments où l'on voit le "~".

L'architecture de ces messages a été définit par le client. Ce schéma représentant l'architecture a été fait dès le début et nous a beaucoup aidé à savoir comment sont hiérarchisés les messages.

