Rapport final : Robot billard

Antoine FONTANEZ - Corentin FROGER - Nathan PIERROT

Tuteur : Pierre-André GUENEGO
Année : 2024/2025

Sommaire

Introduction

Objet du document

Ce rapport final a pour but de documenter le travail que nous avons réalisé durant 7 itérations, pour un total de 270h, dans le cadre du projet Robot Billard.

Présentation du projet

L'objectif de notre projet est de programmer des robots capables de jouer au billard de manière autonome et optimisée. Pour cela nous utilisons une caméra qui permet de récupérer tous les éléments de la scène (robots, boules, table, trous), nous traitons ensuite chaque frame et donnons les ordres aux robots, connectés en réseau. Tout cela est visible sur une interface web que nous avons développée. Afin d'ajouter une plus grande variété de fonctionnalités au projet, nous avons développé en parallèle un simulateur réaliste qui nous permet de tester nos scénarios quand nous n'avons pas accès au matériel physique.

Présentation de l'équipe

Notre équipe est composée de 3 personnes. Bien que nous n'ayons pas défini de rôles stricts, chacun s'est naturellement spécialisé dans certains domaines en fonction de ses compétences et de ses contributions tout au long du projet.

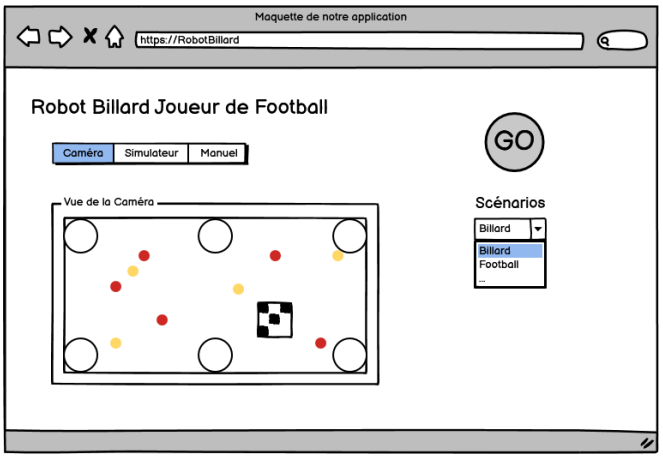
- Antoine FONTANEZ : Développeur du Simulateur et principal contributeur du code du serveur node.js
- Corentin FROGER : Réalisateur du programme C++ du robot et développeur de toutes les fonctions de déplacement des robots
- Nathan PIERROT : Développeur s'étant notamment occupé de l'interface au niveau esthétique mais aussi dans son fonctionnement

Planning

Dans cette partie nous allons passer en revue l'organisation et l'avancement de l'application durant l'étude préalable et les 7 itérations qui l'ont suivi.

Etude préalable

Pour commencer, nous avons réalisé des diagrammes et des maquettes qui nous ont permis de prendre conscience de la complexité du projet et de trouver par la suite des idées et des solutions aux différentes épreuves qui nous attendaient. Nous avons aussi dû faire des choix de conception, ceux qui nous semblaient les meilleurs, par exemple nous avons décidé à ce moment-là de faire notre site web sur une seule page, en changeant juste son contenu en fonction du bouton cliqué.

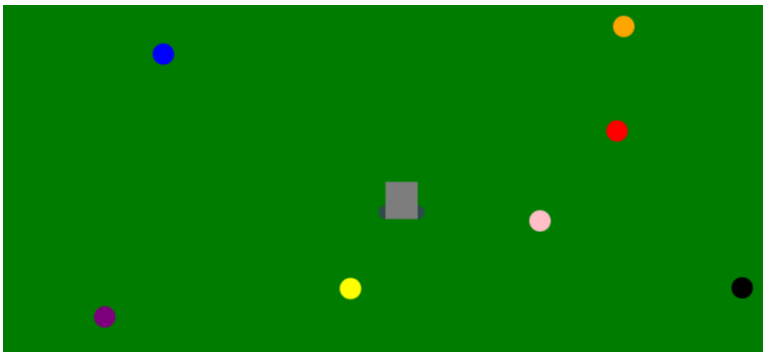


Maquette de l'interface web

Itération 1

Au cours de la première itération nous nous sommes tout d'abord familiarisé avec l'IDE Arduino afin de comprendre le fonctionnement du code du robot pour ensuite pouvoir l'adapter à notre cas. Puis nous avons continué avec un prototype global de l'application, c'est-à-dire un petit serveur fonctionnel, une interface fonctionnelle et un début de simulateur. Dans cette application il était possible d'afficher le flux de la caméra en direct sur un onglet. On a également commencé à utiliser la librairie OpenCV pour détecter et afficher les boules sur le canvas. Ensuite, avec la possibilité de changer d'onglet on pouvait visionner et interagir avec le

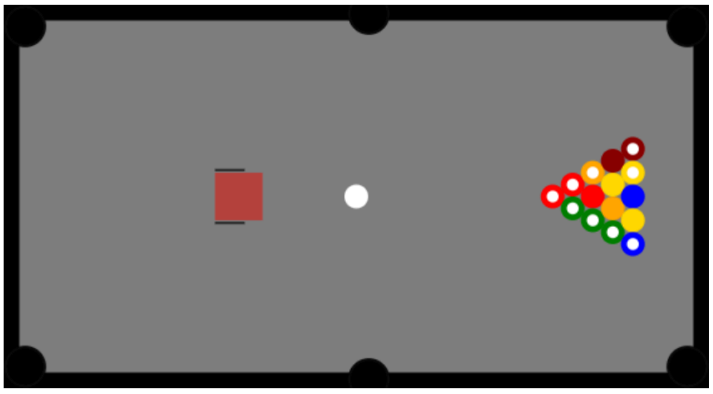
simulateur et ses objets et enfin sur le dernier onglet on pouvait faire bouger manuellement le robot réel en appuyant sur des boutons de la page. Le serveur a permis de réaliser cette fonctionnalité car il sert de passerelle entre le robot et la page web, ceci est possible car tous les éléments sont connectés au même réseau.



Première version de notre simulateur

Itération 2

Lors de la seconde itération, nous avons déjà effectué un refactoring de notre code afin de permettre une meilleure évolution. Puis nous avons continué à utiliser la librairie OpenCV pour cette fois-ci détecter les ArUco. Nous avons fait le choix d'en mettre aux quatre coins de la table de billard pour pouvoir déterminer ses dimensions sur le canvas. De plus on a traité les clics sur le canvas en récupérant leur position, cela nous a aidé à tester une autre fonctionnalité que nous avons programmé : le calcul de distance entre 2 points du canvas. En parallèle de ce travail sur la caméra, nous avons travaillé sur le simulateur en le rendant plus réaliste visuellement (couleurs, tailles, …) et en ajoutant de nouvelles configurations pré faites (on avait déjà une configuration random et on a rajouté une configuration simple, de billard et de foot).



Deuxième version de notre simulateur (presque identique à la version actuelle)

Itération 3

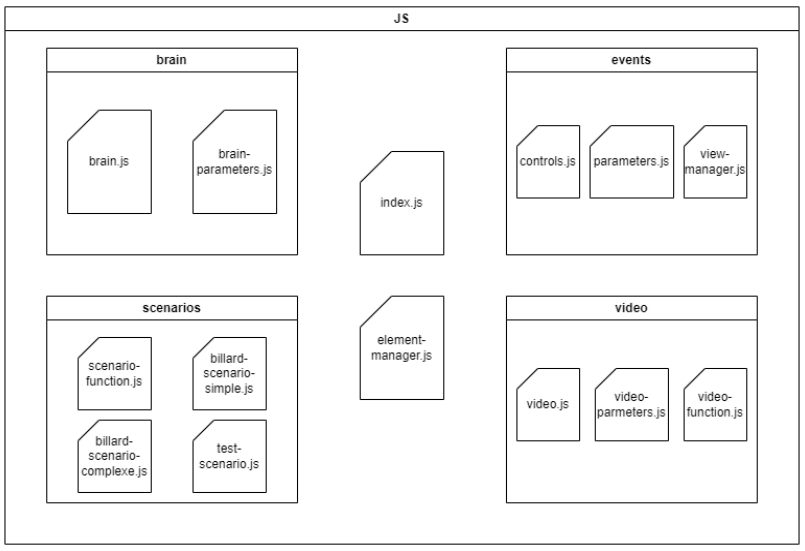
En ce qui concerne cette itération, nous avons grandement réutilisé les fonctionnalités programmées dans l'itération précédente pour les améliorer et les pousser plus loin. Nous avons par exemple fait en sorte que, lors d'un clic sur le canvas, le robot réel se déplace à la position cliquée. Nous avons également amélioré la détection des ArUco pour obtenir leur orientation et ainsi connaître celle du robot. Et toujours avec les ArUco, nous avons différencié ceux des coins de la table de celui du robot. Cela nous a permis d'ignorer les boules détectées en dehors de la table (erreurs de détections, mais qui sont de toute façon inatteignables pour le robot). Puis nous avons fait en sorte de permettre à l'utilisateur de choisir le robot à contrôler. Nous avons aussi amélioré le simulateur en utilisant la détection de cercle directement dedans et nous avons pu commencer à programmer un scénario de test afin de nous préparer à en faire de plus complexes par la suite.

Détection des boules dans le simulateur

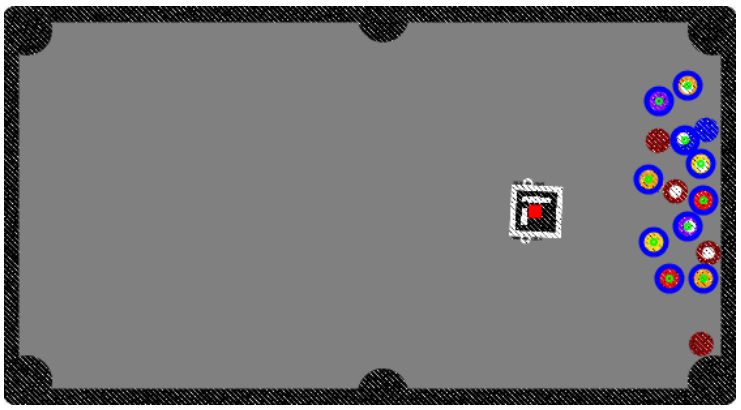
Itération 4

Lors de cette itération, nous avons grandement amélioré notre scénario, le robot est maintenant capable de détecter la boule la plus proche et de la pousser jusqu'à ce qu'elle tombe dans un trou (pour l'instant il ne fait que la pousser sans prendre en compte la position des trous). Ce scénario est possible à la fois dans le simulateur et sur la table réelle. Concernant le simulateur, nous avons tenté de le rendre encore plus réaliste, nous avons donc ajouté du bruit pour brouiller la détection avec OpenCV et simuler les réelles interférences de la caméra. De plus, nous avons aussi ajouté la détection d'ArUco pour se rapprocher au maximum de la structure réelle du projet. Nous avons également basé la reconnaissance des trous de la table de billard sur les positions des ArUcos plutôt qu'en se basant sur la distance des cercles avec la bordure de la table.

Tout ce code commençait à devenir complexe à comprendre, nous avons donc refactorisé le fichier index.js et refait toute la structure des fichiers js (image explicative ci dessous). En plus de cela nous avons amélioré "l'intelligence" du robot, il est maintenant capable de se déplacer de manière plus fluide grâce à des mouvements en courbe. Et enfin, concernant l'expérience utilisateur, nous avons laissé le choix d'afficher ou non les traitements de OpenCV sur le canvas et avons développé une petite interface mobile pour permettre à n'importe qui de contrôler facilement les robots à distance.



Structure d'une partie des fichiers de notre application



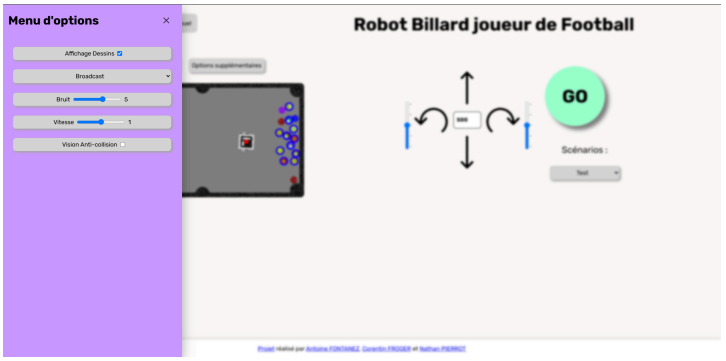
Simulateur réaliste avec du bruit et la détection d'ArUco

Itération 5

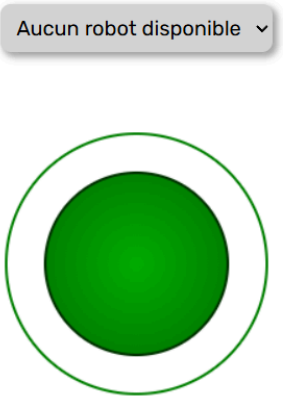
Durant cette itération, nous avons notamment créé un scénario complexe, qui est pareil qu'un scénario "simple", hormis que le robot cherche à faire rentrer les balles dans le trou le plus proche plutôt que de simplement foncer dedans. Pour cela, il cherche la balle la plus proche de lui, puis le trou le plus proche de la balle, et s'aligne de manière à pouvoir pousser la balle directement dans le trou.

Nous avons également fait en sorte d'actualiser en direct la liste des robots, de sorte à voir en permanence les robots disponibles. Nous avons également ajouté un menu d'options, pour pouvoir aérer la page et avoir une meilleure visibilité, nous avons également fait en sorte de pouvoir arrêter un scénario en cours plutôt que celui-ci continue à l'infini.

Nous avons également ajouté un joystick à l'interface mobile pour pouvoir contrôler le robot d'une manière plus intuitive.



Menu d'option du simulateur



Vue depuis un téléphone

Itération 6

Lors de cette itération, nous avons fait en sorte de pouvoir envoyer des ordres à différents robots, sans qu'ils soient envoyés avec pour référence le même robot, ainsi, ils ne reçoivent plus les mêmes ordres mais ceux-ci sont bel et bien personnalisés pour chacun d'entre eux. Nous avons également fait en sorte d'empêcher les robots de rentrer dans un trou, ainsi qu'afficher les ID des ArUcos plutôt que leur adresse IP.

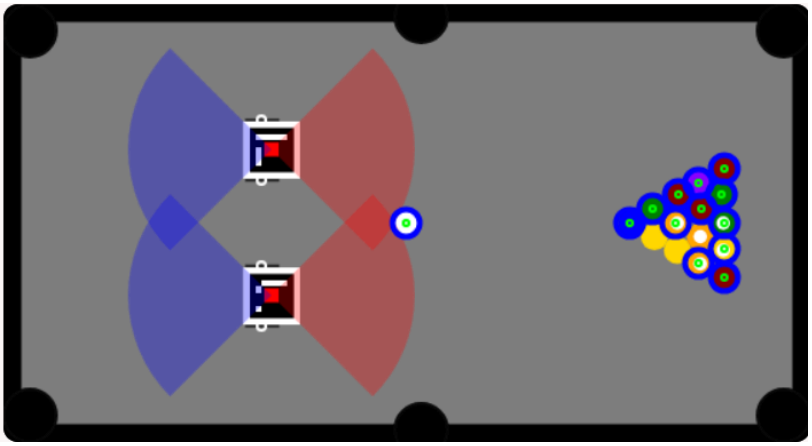
De plus, nous avons fait en sorte que les robots démarrent leurs moteurs de manière progressive, ce qui permet d'éviter qu'ils se lèvent subitement lorsqu'on leur demande de démarrer leurs moteurs à une vitesse élevée. Nous avons également créé un scénario collaboratif, où plusieurs robots sont capables de jouer au billard, tout en évitant les collisions entre eux.

Id de l'ArUco d'un robot, visible depuis la vue Caméra

Itération 7

Lors de cette itération, nous avons surtout corrigé les différents bugs connus, et ajouté des fonctionnalités simples qui ne risquent pas de prendre trop de temps à ajouter et qui ne risquent pas de causer trop de bugs.

Par exemple, nous avons fait en sorte qu'il soit possible de contrôler les robots depuis plusieurs interfaces, ainsi, on peut imaginer un robot qui joue de manière intelligente et un autre qui est contrôlé manuellement depuis l'interface mobile. Nous avons également fait en sorte d'afficher la détection anti-collisions des robots, à la fois dans le simulateur et dans la vraie vie.

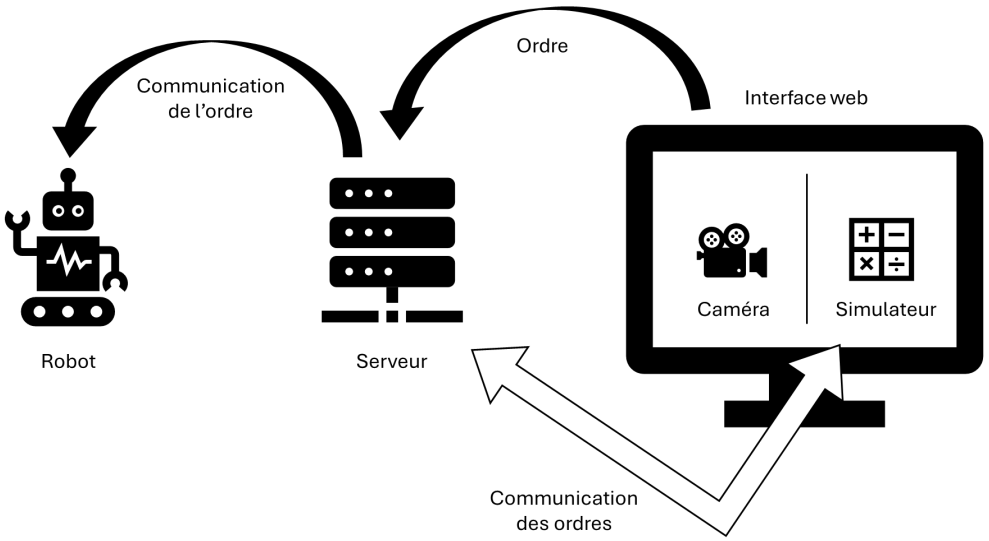


Angle d'anti-collision des robots (en rouge à l'avant et en bleu à l'arrière)

Analyse

Structure du projet

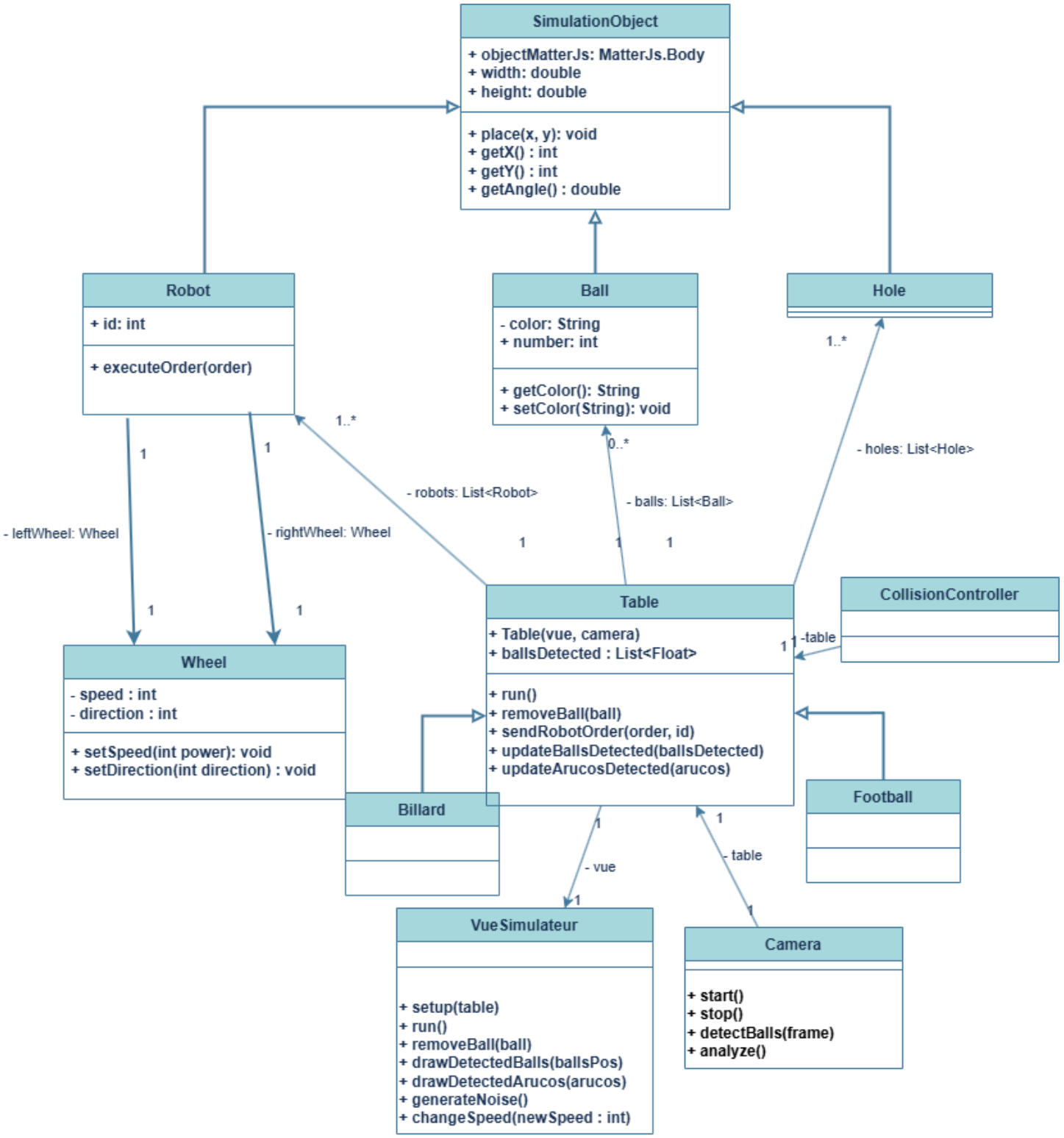
Notre projet se compose de plusieurs parties de programmation, nous avons une partie arduino pour le code interne du robot qui lui permet de recevoir des ordres et de les exécuter. Ensuite, nous avons le serveur Node.js qui permet la connexion et la communication des ordres entre le robot et l'interface web. Cette interface web est la plus grosse partie du projet et est composée de plusieurs sous parties. La première se compose d'une vue de la caméra qui permet de visualiser et de traiter le flux vidéo de la caméra branchée à l'ordinateur. Cette caméra filme l'entièreté de la table de billard et toutes ses boules. La seconde comprend la réalisation d'un simulateur réaliste programmé grâce à la librairie Matter.js qui nous permet d'obtenir une physique réaliste. Ce simulateur reçoit également des requêtes du serveur pour faire exécuter des ordres au robot. Voici ci-dessous un bref schéma récapitulatif de notre structure :



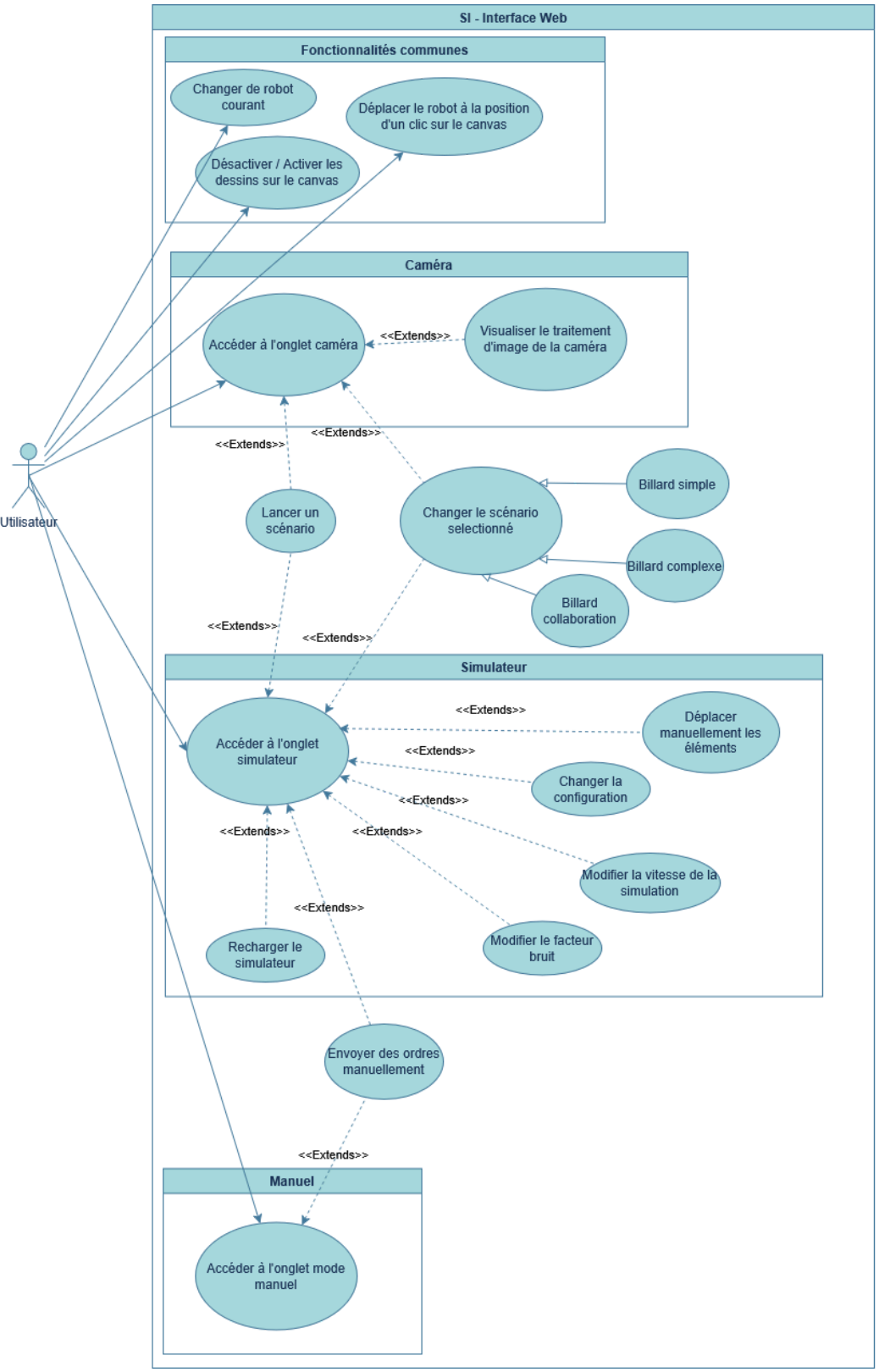
Diagrammes UML

Afin de rendre nos explications le plus claires possible, voici quelques diagrammes UML de notre structure :

- Diagramme de classe :



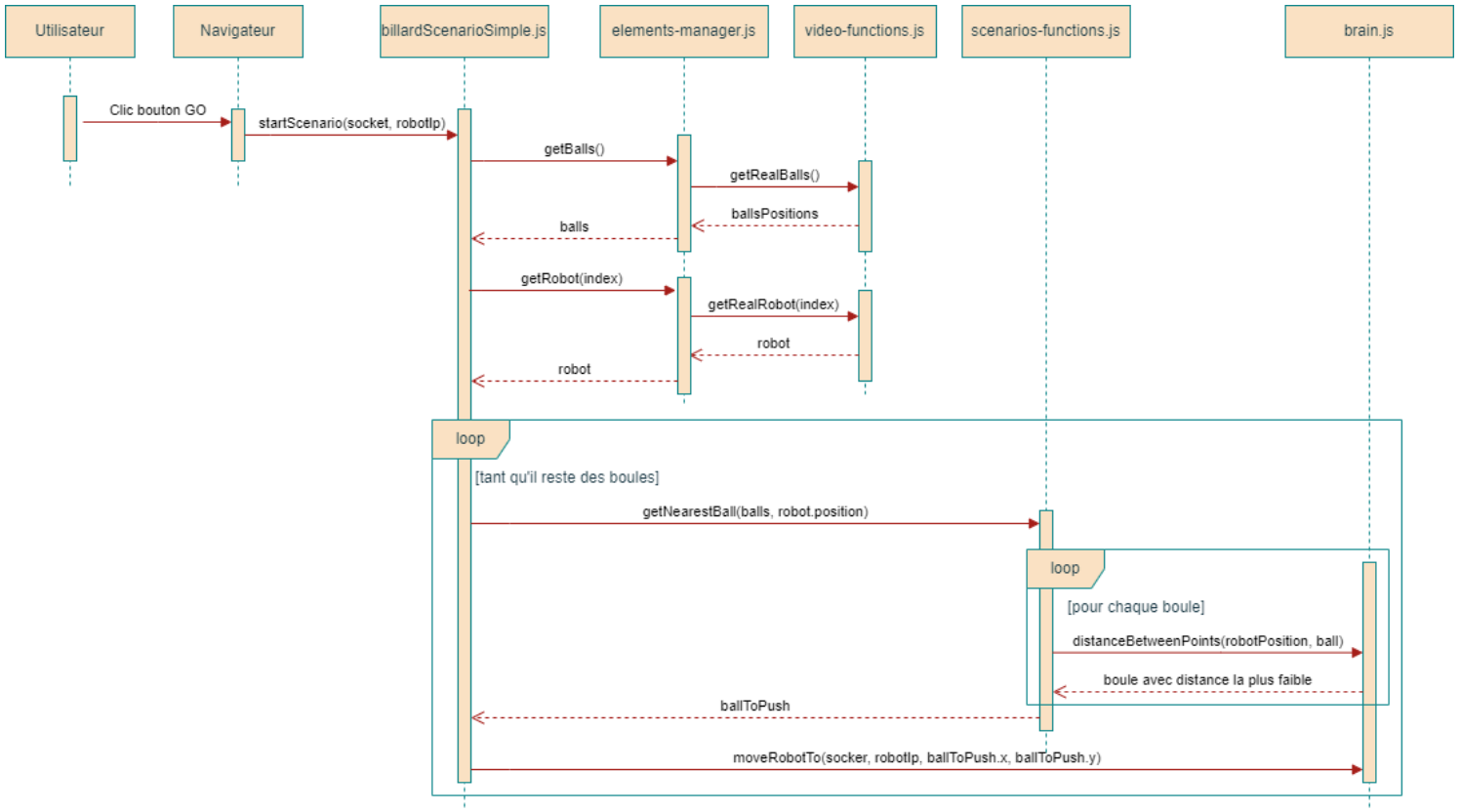
- Diagramme de CU :



Précision

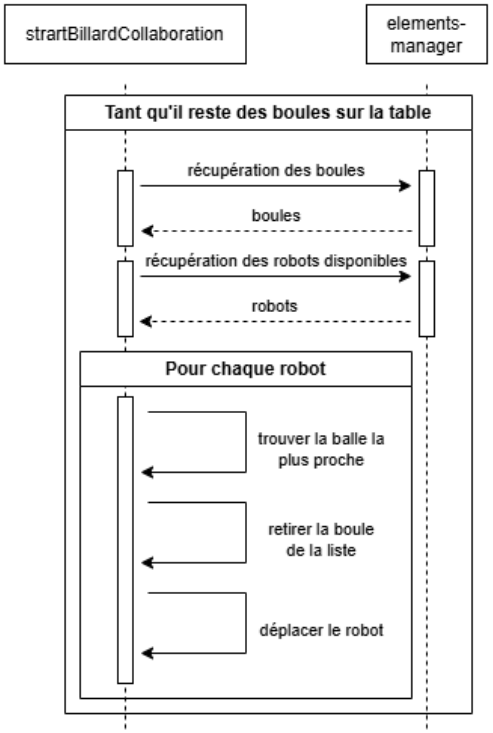
Les fonctionnalités communes sont accessibles depuis n'importe quel onglet

- Diagrammes de séquences :
  (Scénario billard simple)



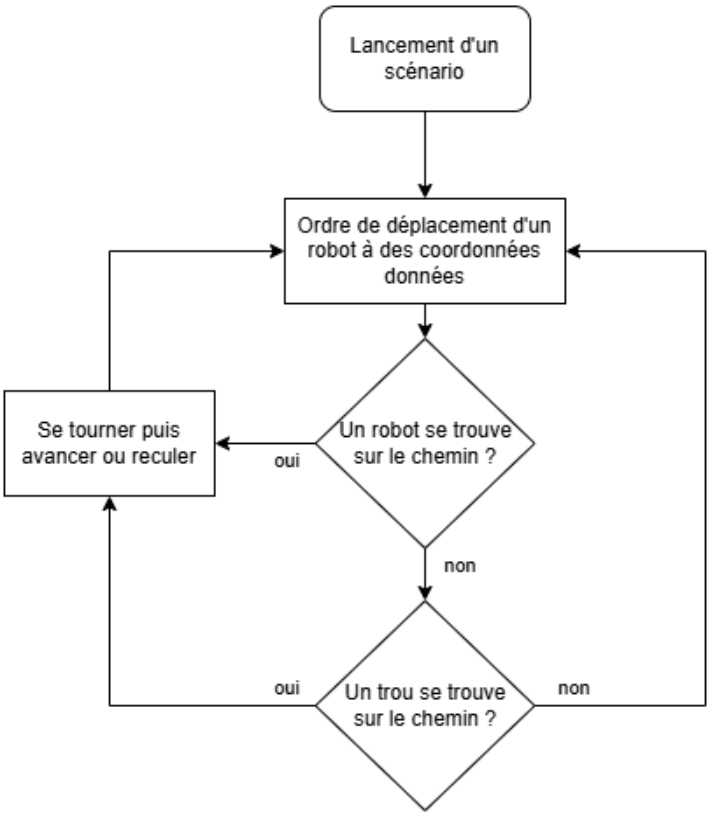
Ce diagramme de séquence montre le déroulement d'une partie de billard simple (le robot se contente d'aller vers la boule la plus proche), lorsque l'utilisateur clique sur le bouton GO en ayant choisi ce scénario, le scénario se lance et cherche à obtenir la position des boules et du robot détectées par OpenCV, ensuite, tant qu'il reste des boules à mettre dans des trous, on cherche à savoir quelle boule est la plus proche du robot, puis on demande au robot de s'y déplacer.

- Diagramme simplifié du scénario collaboration :



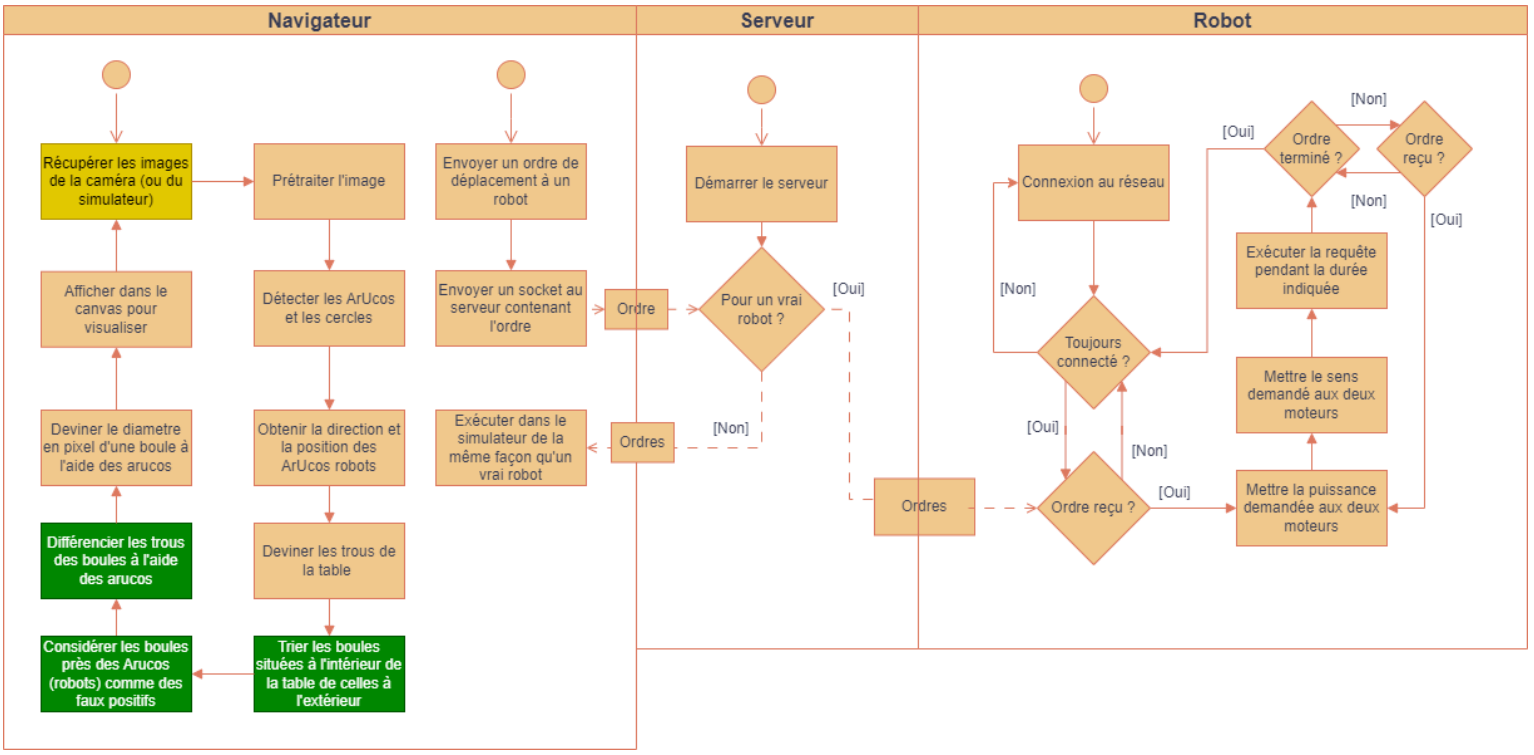
Tout comme le scénario simple, chaque robot cherche à se déplacer vers la boule la plus proche. Mais dans ce scénario, chaque robot cible une boule différente, une même boule ne peut pas être la cible de plusieurs robots en même temps. Pour cela, nous retirons la boule choisie de la liste de boules afin que les autres robots n’aient plus ce choix.

- Diagramme représentant les priorités d'actions des robots dans leurs déplacement :



Ce diagramme explique l'ordre de priorité lors du déplacement d'un robot. Un robot va tout d'abord regarder si un autre robot ne se trouve pas sur son chemin avant de regarder si un trou ne se trouve pas sur son chemin. Dans le cas où un obstacle obstrue son chemin, il va faire un détour afin de l'éviter. Puis il va repartir tout droit vers sa destination, toujours en vérifiant les autres robots et les différents trous.

- Diagramme d’activités :



Ce diagramme explique le fonctionnement général de l’application dans son état actuel. Une fois le serveur Node.js lancé, celui-ci vérifie en permanence si un ordre doit être envoyé à un robot, et s'il doit l’envoyer à un vrai ou à l’un de ceux du simulateur.

Dans le navigateur, côté client, on utilise OpenCV pour récupérer les images de la caméra, il faut ensuite traiter l’image pour mieux détecter les objets (boules, ArUcos, trous), on stocke ensuite leur position, et l’orientation des ArUcos robots. On peut également connaître la zone du billard à l’aide des ArUcos et donc pouvoir considérer les boules situées à l’extérieur de la table comme des faux positifs, nous utilisons également les ArUcos pour différencier les trous de table des boules.

Les robots quand à eux doivent d’abord se connecter au réseau (ils se reconnectent si besoin) et exécutent les ordres que le serveur leur envoi, après quoi ils ajustent la vitesse et le sens des moteurs concernés et exécutent la requête pendant la durée indiqué, sachant que s’ils reçoivent un nouvel ordre, ils l’éxécutent à la place de l’ancien s’il n’était pas terminé.

Evolution par rapport à l’étude préalable

Les diagrammes créés lors de l’étude préalable ont été amenés à changer plus ou moins fortement selon les cas, dans le cas du diagramme des cas d’utilisations, nous avons au final abandonné l'idée des fonctionnalités d’import et de sauvegarde de configurations pour le simulateur, car jugée inutile. Nous avons en revanche ajouté des fonctionnalitées auxquelles nous n’avions pas pensé à la base, comme pouvoir directement détecter des formes à l’aide d’OpenCV dans le simulateur, et de rajouter du “bruit”, toujours directement dans le simulateur pour pouvoir dégrader la détection d’OpenCV comme dans la réalité.

La façon dont nous avons conceptualisé les classes dans le simulateur a également été modifiée, nous avons rendu le MVC plus robuste, nous avions à la base pensé que nous devions faire une classe Serveur pour simuler le serveur dans le simulateur, mais nous avons simplement réutilisé le serveur préexistant.

Nous avons également ajouté une classe caméra gérant la détection OpenCV dans le simulateur, ce qui n’était pas prévu à la base, mais qui s’est avéré très utile pour pouvoir tester nos algorithmes et scénarios plus facilement.

Dans le cas du diagramme de séquence détaillant un scénario de billard, il s’est avéré assez proche de la façon dont il a été réalisé. En soit, on récupère les positions des boules et des robots détectés par OpenCV (réels ou ceux du simulateur), on cherche à savoir quelle est la boule la plus proche du robot concerné, puis on lui indique de s’y déplacer.

Patrons de conceptions

Dans notre code final, nous avons appliqué plusieurs patrons de conception afin d'assurer une architecture claire et facilement maintenable. Les principaux patrons implémentés sont les suivants :

1. MVC

Nous avons structuré notre simulateur selon le patron de conception MVC. Ce choix permet de séparer les responsabilités entre les différentes couches de l'application.

Cette architecture permet d'ajouter ou de modifier des éléments dans l'interface ou dans la logique sans impacter l'ensemble du code, ce qui pourrait faciliter la maintenance et les évolutions futures du projet.

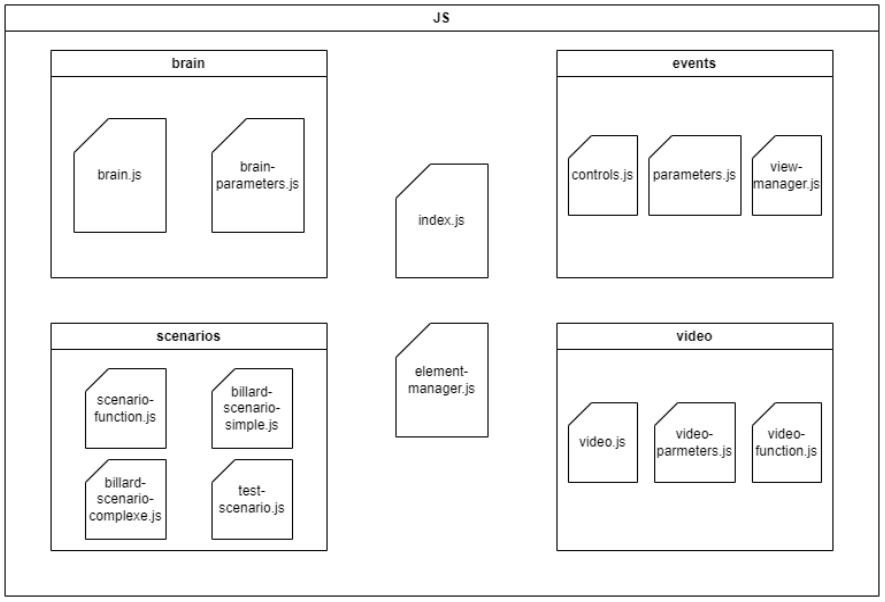
2. Stratégie

Quelque chose de légèrement similaire à un patron stratégie a été réalisé dans le cadre de la gestion des différents scénarios. Chaque script de scénario possède une fonction startBillardScenario qui prend les mêmes paramètres : la socket, et l'id du robot concerné. Concrètement, l'exécution d'un scénario se fait à partir d'un switch basé sur le nom du scénario sélectionné. Chaque cas dans ce switch instancie une classe de scénario spécifique, puis appelle sa méthode startBillardScenario.

Bien que cette approche ne respecte pas tout à fait les principes du patron Stratégie (notamment en ce qui concerne l'usage d'une interface commune et le recours au polymorphisme), cela rend l'ajout de nouveaux scénarios relativement simple, tant qu'ils respectent le format de fonction attendu.

Réalisation

Architecture logicielle



Voici la structure de nos fichiers js, tout d'abord, nous retrouvons au centre le fichier *index.js* qui permet de faire le lien entre tous les fichiers js et le fichier *index.html*. Au même niveau, nous avons placé le fichier *element-manager.js* qui, lui, regroupe toutes les fonctions de récupération d'éléments communs aux différentes vues, par exemple la récupération des différentes boules de la table, que ce soit sur le billard réel ou sur le simulateur.

Ensuite, nous retrouvons en haut à gauche de l'image un dossier *brain* qui contient tous les fichiers utiles à "l'intelligence" du robot. Notamment le fichier *brain.js* qui contient entre autres les méthodes de déplacement et le fichier *brain-parameters.js* que nous utilisons pour regrouper toutes les constantes utilisées par le fichier *brain.js*. Cela nous permet de modifier plus facilement différents paramètres de déplacement sans avoir à rechercher toutes les occurrences dans le code.

Puis, en haut à droite, nous avons le dossier event, chargé de regrouper tous les fichiers en lien avec les événements et les actions réalisables par l'utilisateur :

- Pour commencer *view-manager.js* s'occupe de gérer le changement de la vue lorsque l'utilisateur décide de se déplacer.

- Le fichier *controls.js* est responsable des actions liées aux boutons de déplacements manuels (les flèches directionnelles).
- Et pour finir le fichier *parameters.js* s'occupe des boutons de configuration du canvas : activer ou non les dessins de OpenCV, le bruit, changer la vitesse de simulation…

Le troisième dossier, *scenarios*, a été créé pour accueillir les différents scénarios et les méthodes communes à l'exécution de ceux que nous avons créés. Avoir un dossier spécial nous permet d'éviter au maximum la redondance et de créer de nouveaux scénarios plus facilement en utilisant les méthodes déjà existantes.

Et en dernier, nous retrouvons un dossier conçu spécialement pour le traitement des canvas avec OpenCV, il contient tout ce dont nous avons besoin pour traiter la caméra comme le simulateur.

Tests

Etant donné que notre projet touche à des éléments physiques, il est difficile de tester nos algorithmes en faisant de simples tests unitaires. Cependant nous avons tout de même pu réaliser des tests empiriques afin de reproduire au mieux les conditions réelles dans notre simulateur. Ainsi, afin de faire nos tests nous avons utilisé le simulateur que nous avons développé à cet effet.

Difficultés rencontrées

Apprendre à programmer l'Arduino et coder en C++ nous a posé de nombreux problèmes au début du projet, par exemple faire en sorte que l'Arduino soit connecté au serveur dès qu'il est alimenté, faire en sorte que l'arduino soit capable d'utiliser les moteurs qui lui sont branchés, ou encore utiliser des JSON en C++.

Nous avons également rencontré des problèmes lorsque nous testions le mode manuel du robot réel et celui dans le simulateur, en effet, certains des robots réels n'ont pas leurs deux moteurs dans le même sens que les autres, or, le code de l'arduino part du principe que ce n'est pas le cas, nous avons donc dû inverser le sens des moteurs dans le code Arduino et mit de côté les robots avec les moteurs inversés.

Dans le simulateur cette fois, nous avions rencontré un bug lorsque nous voulions faire se déplacer le robot à un endroit donné, en effet, le robot avait d'abord besoin de se tourner vers sa cible avant d'avancer, or, il arrivait que le robot tourne dans le sens le moins optimal possible, il arrivait dans le pire des cas, qu'il fasse un tour complet sur lui-même plutôt que de simplement tourner un tout petit peu dans l'autre sens. Il s'est avéré que cela était dû à la librairie Matter.js qui représentait les angles de moins l'infini à plus l'infini, et non pas dans une fourchette précise (par exemple de 0° à 360°), ce bug nous a pris beaucoup de temps et de ressources à corriger, mais nous avons réussit à trouver sa cause et à trouver une solution.

Lors de l'itération 6, lorsque nous avons voulu rendre le démarrage des moteurs progressif, il pouvait arriver que lorsque l'on téléversait un programme sur un Arduino, les moteurs n'étaient plus détectés comme correctement câblés. Nous avions pourtant bien pris soin de vérifier les installations des robots, mais rien à faire, ce problème pouvait persister plusieurs heures, ce qui a grandement ralenti les tests.

Nous avons également rencontré des problèmes physiques avec les robots. Parfois, il y avait des faux contacts avec les câbles, ou même des câbles qui se débranchaient, ce qui nous obligeait à aller les remettre pour continuer les tests.

Répartition du travail

Notre équipe est composée de 3 personnes, ce qui rend l'organisation plus facile, chacun peut facilement trouver quelque chose à faire sans empiéter sur le travail des autres. Nous avons donc décidé de diviser le projet en 3 "sous projets" et chacun peut travailler sur sa partie, nous avons donc dans les grandes lignes : Antoine FONTANEZ qui s'est principalement occupé de programmer le simulateur, Corentin FROGER qui à codé la détection d'OpenCV et l'Arduino, et Nathan PIERROT a fait le frontend en plus d'un soutien à Corentin.

Le tableau recense les principales fonctionnalités réalisées à chaque itérations et les personnes qui se sont investies dans le développement de chaque fonctionnalité :

Fonctionnalités réalisées	**Antoine FONTANEZ**	**Corentin FROGER**	**Nathan PIERROT**
Itération 1			
Prototype de l'interface web			X
Prototype du serveur Node.js	X		
Prototype du simulateur	X		
Connecter le robot au même serveur que le navigateur	X	X	X
Gérer le flux de la caméra dans le navigateur		X	X
Contrôler les robots manuellement	X	X	
Reconnaître les boules avec la librairie OpenCV		X	X
Afficher et bouger des objets dans le	X		

simulateur			
Itération 2			
Refactoring HTML/js, serveur Node.js et Arduino	X	X	X
Afficher le simulateur de façon plus réaliste	X		
Détecter la position des clics sur le canvas	X		X
Calculer la distance entre 2 points sur le canvas		X	
Reconnaître les ArUco avec OpenCV		X	
Ajout d'une page de chargement			X
Ajouter des configurations pré-faites pour le simulateur		X	X
Itération 3			
Être capable de déplacer le robot à un endroit précis (grâce à un clic sur le canvas)	X	X	X
Envoyer des ordres différents à différents robots	X	X	X
Exécuter un ordre pendant un certain temps ou jusqu'à la réception d'un nouvel ordre			X
Utiliser la détection de cercles dans le simulateur	X		
Programmer un scénario de test dans		X	

le simulateur			
Différencier les ArUco des robots et ceux des coins du billard	X		
Obtenir l'orientation du robot grâce à ArUco		X	
Ignorer les boules détectées en dehors de la table		X	
Itération 4			
Programmer un scénario de billard simple		X	X
Détection d'ArUco dans le simulateur	X		
Ajouter du "bruit" au simulateur	X		
Considérer les cercles détectés sur un ArUco comme de faux positifs	X	X	
Pouvoir modifier la vitesse de la simulation	X		
Calcul des trajectoires et de la vitesse du robot pour qu'il puisse bouger de manière fluide	X	X	X
Refactoring index.js et la structure des fichiers js	X	X	X
Faire en sorte de voir la vidéo avec et sans les dessins			X
Interface mobile simple pour contrôler les robots réels		X	

Itération 5			
Scénario complexe	X		
Actualisation de la liste des robots en direct			X
Ajout d'un menu d'options			X
possibilité d'arrêter un scénario en cours	X		
Amélioration de l'interface mobile		X	
Itération 6			
Envoyer des ordres différents aux différents robots plutôt que de les envoyer en se basant sur le même robot	X		
Afficher les ID des ArUcos plutôt que les IP des robots	X		
Scénario billard collaboratif			X
Démarrage progressif des roues des robots		X	
Empêcher le blocage des robots dans les trous		X	X
Prototype d'anti collision entre les robots	X		
Itération 7			
Affichage de la zone d'anti-collision entre les robots	X		X
Optimisation de la navigation dans l'interface			X

Permettre la connexion de plusieurs interfaces	X		
Perfectionner l'anti-collision	X	X	
Amélioration de la fonctionnalité : empêcher robot de se bloquer dans un trou	X	X	

Conclusion

Ce que ce projet nous a apporté

Ce projet nous a apporté de nombreuses connaissances en robotique grâce à la programmation Arduino, nous avons également gagné en compétences au niveau du serveur que nous avons mis en place avec Node.js et les websockets. L'utilisation d'OpenCv pour la détection d'objets tels que les ArUcos et les cercles nous a permis de découvrir le fonctionnement de cette librairie très utilisée dans la robotique. Nous avons utilisé une autre librairie que nous ne connaissions pas : Matter.js, utilisée pour simuler une physique réaliste dans le simulateur.

Reprise de notre projet

Nous ne pensons pas qu'il serait judicieux, pour d'autres étudiants l'année prochaine, de reprendre et continuer notre projet. Nous avons déjà réalisé le plus intéressant du projet : programmer les robots, des scénarios et le simulateur réaliste. Tout cela nous a beaucoup appris au niveau robotique et réseau.

De plus, afin de vraiment comprendre le travail que nous avons déjà fait, il est plus pertinent de recommencer tout depuis le départ que de juste essayer de comprendre notre code. Nous ne sommes pas des développeurs professionnels et certaines parties du code ne sont pas très bien optimisées et organisées (même si nous avons fait de notre mieux pour avoir un code clair).

Annexes

Mode d'emploi pour l'installation

1) Récupérez le projet git :
   `git clone https://github.com/Akip2/RobotBillard.git`

2) Déplacez-vous au répertoire source du projet
   `cd RobotBillard/production/`

3) Installez les dépendances
   `npm install`

Mode d'emploi pour l'utilisation

Étapes de lancement

1) Branchez la borne wifi sur le port ethernet de votre machine et connectez votre machine au réseau Eduroam

2) Branchez la caméra sur votre machine

3) Déplacez-vous au répertoire source du projet
   `cd RobotBillard/production/`

4) Lancez le serveur
   `node server.js`

5) Recherchez "localhost:8001" dans la barre de recherche d'un navigateur afin d'accéder à l'application

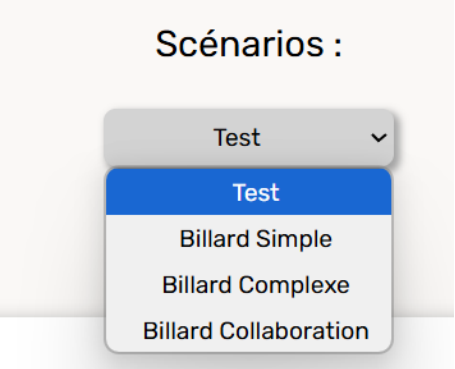
Utilisation

Une fois connecté à l'application, trois onglets sont à votre disposition :

1) **Caméra** : Permet de faire jouer des scénarios au(x) robot(s) réel(s).

   Pour jouer un scénario il vous suffit de sélectionner le robot concerné (Broadcast pour tous)

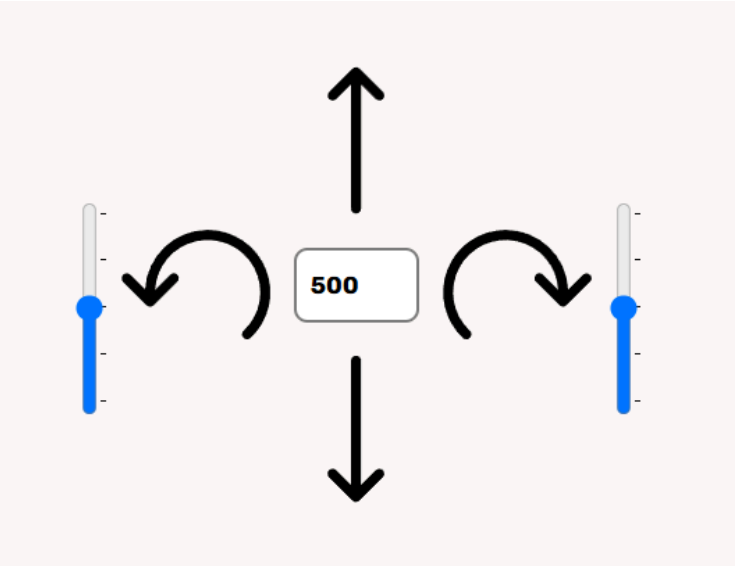
   Sélectionner le scénario que vous voulez jouer :



   - **Test** : le robot se déplace en formant un carré
   - **Billard Simple** : le robot fonce sur la boule la plus proche de lui
   - **Billard Complexe** : le robot se déplace derrière la boule la plus proche de lui en s'alignant avec un trou, puis la pousse pour tenter de la rapprocher du trou
   - **Billard Collaboration** : même principe que billard simple, mais s'il y a plusieurs robots, ils ciblent des boules différentes

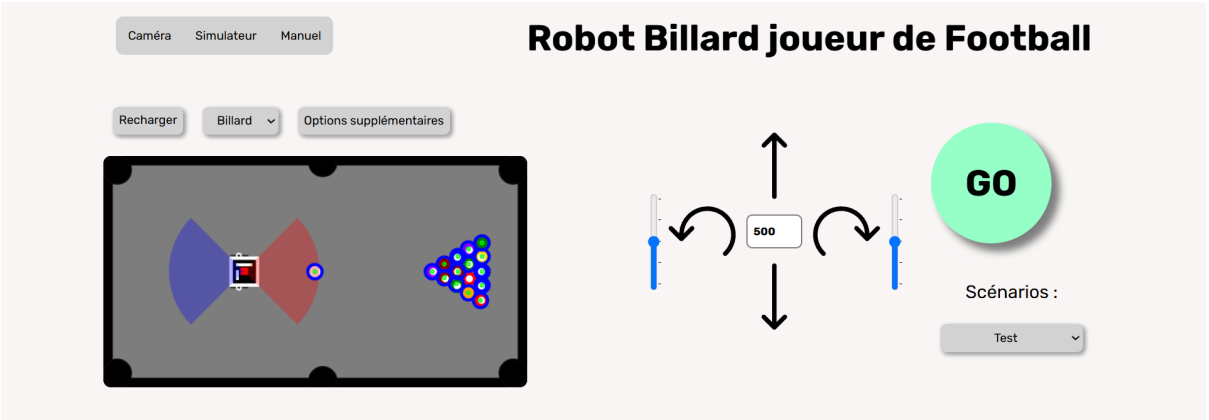
   Une fois le scénario choisi, lancez le scénario en cliquant sur le bouton GO

2) **Manuel** : Permet de bouger manuellement le ou les robots réels, en utilisant le menu de boutons flèches

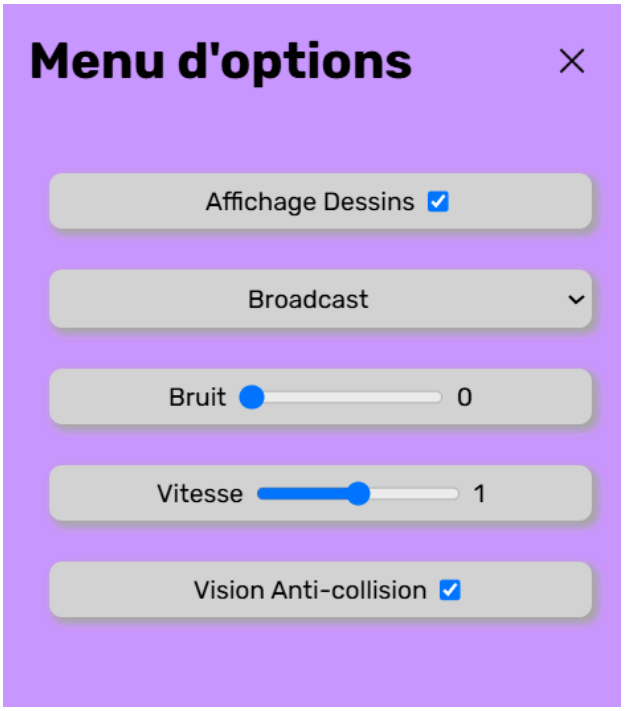


- Les **sliders aux extrémités** permettent d'ajuster la puissance des deux moteurs (droite et gauche).
- La **valeur au centre** correspond à la durée en milliseconde durant laquelle l'ordre sera exécuté par le robot
- Les **boutons en forme de flèches** dirigent le robot dans une direction spécifique (en avant, en arrière, tourner sur lui-même à droite, tourner sur lui-même à gauche)

3) **Simulateur** : Permet de tester des algorithmes sans utiliser les robots réels. Il est possible de jouer des scénarios et de tester des configurations particulières



Un certains nombre de paramètres vous sont accessibles pour tester le simulateur dans différentes conditions (accessibles via le bouton “Options supplémentaires”) :



- **Afficher Dessins** : activer ou non les dessins de debug
- **Sélection du robot courant** : choisir le robot concerné, sélectionner “Broadcast” pour tous
- **Bruit** : ajuster le niveau de perturbation visuelle, afin de simuler une caméra de mauvaise qualité
- **Vitesse** : accélérer ou ralentir l'exécution du simulateur
- **Vison Anti-collision** : activer ou non la visualisation des zones d’anti-collision

Comme pour la page caméra, vous pouvez lancer un scénario avec le bouton GO. Et comme pour la page manuel, vous pouvez déplacer manuellement le robot avec les boutons flèches.

Bugs connus

Voici une liste non exhaustive des bugs que nous savons présents dans notre application :

- Les robots peuvent essayer de traverser la bordure de la table de billard lorsqu'ils essaient de se déplacer quelque part, ce qui fait qu'ils se coincent puisqu'ils ne peuvent pas traverser la bordure.
- L'application rencontre des problèmes lorsqu'il n'y a pas les ArUcos des coins de la table.
- Le démarrage progressif des roues peut entraîner des mouvements non voulus des robots, par exemple, il se peut que le robot n'aille pas en ligne droite lorsqu'il commence à se déplacer après un long moment d'inactivité.
- Nous avons remarqué que certaines boules avaient du mal à être détectées, lorsque nous avons ajouté un bouton pour afficher l'image prétraitée par OpenCV, (celle directement utilisée pour détecter des cercles et autres) nous avons compris que cela était dû au fait que nous convertissons l'image en échelle de gris, or, certaines couleurs se confondaient avec le gris de la table.

Idées d'amélioration

Notre application comporte des fonctionnalités pouvant être améliorées, par exemple :

- Lorsque la liste des ID des robots disponibles s'actualise, si le robot sélectionné se déconnecte et se reconnecte immédiatement, le navigateur passera automatiquement au Broadcast, ce qui rend difficile le contrôle de robots précis, par exemple depuis un téléphone.
- Le scénario collaboratif est identique au scénario simple, hormis le fait que 2 robots ne peuvent pas prendre pour cible la même boule. On pourrait par exemple faire qu'ils se fassent des passes si on souhaite faire un réel scénario collaboratif.
- Le joystick est certes simple à utiliser car il n'y a besoin que d'un doigt pour contrôler un robot, mais le tourner peut être compliqué car il risque d'avancer en même temps. On pourrait trouver un moyen de "bloquer" certaines directions pour faciliter le fait de tourner un robot.
- Nous pourrions améliorer le prétraitement des images, par exemple ajouter du flou, retirer les ombres, ou d'autres filtres pour améliorer la détection de cercles et d'OpenCV en général.