RAPPORT DU PROJET :

Communication entre plusieurs robots Dobots Magician par Arduino

Professeur référent : Mr. BOIMOND Jean-Louis

Auteurs du projet :

- CHOLET Byron
- PAQUEREAU--GASNIER Alexis

Année : 2022-2023

université angers

POLYTECH ANGERS

SOMMAIRE

I. REMERCIEMENTS

Avant de commencer ce rapport, nous souhaitons remercier quelques personnes spéciales qui nous ont accompagné tout au long de notre projet.

- Mr. BOIMOND Jean-Louis, notre professeur référent de projet, qui a toujours su être présent pour nous lorsque nous avions besoins de conseils ou de matériel. Il n'a pas hésité à montrer son engouement pour notre projet et nous proposer des améliorations auxquelles nous n'aurions pas pensé seuls.

- Mr. MORILLE Adrien et Mr. LAMBERT Maxime, étudiants en PeiP 2, avec qui nous avons régulièrement partagé l'avancée de notre projet. Ils nous ont aidé à plusieurs reprises en nous conseillant sur la partie électronique du projet, notamment en prenant de leur temps pour nous partager leur expérience sur le logiciel EAGLE.

- Mr. LAGRANGE Sébastien et Mr. VERRON Sylvain, pour ne pas avoir hésité à nous aider lorsque deux élèves de PeiP 2 dont ils n'étaient pas responsables leur ont demandé leurs services.

Nous tenons également à remercier tout le corps enseignant de Polytech Angers, ainsi que les personnes qui travaillent dans l'ombre pour rendre la réalisation de ces merveilleux projets possible. Ce fut une expérience exceptionnelle et nous sommes heureux d'avoir eu la chance d'y participer.

II. RESUME / ABSTRACT

A. Version Française

Il s'agissait de notre premier projet à grande échelle, et les défis que nous avons dû surmonter étaient de taille. Nous sommes deux élèves en deuxième année de classe préparatoire à Polytech Angers. En binôme, nous avons non seulement créé un outil Arduino pour permettre la communication entre plusieurs robots Dobot Magician, mais aussi démontré les possibilités derrière une telle collaboration grâce à la construction d'une ville miniature !

Dans ce rapport, nous revenons sur toutes les étapes clés des six derniers mois passés sur ce projet. L'adoption de robots éducatifs a été accompagnée par la découverte du C++ et de la programmation sur Arduino. En partant de peu, notre ambition nous a poussé à créer une bibliothèque open-source pour permettre le contrôle de plusieurs robots depuis une unique carte Arduino. S'en sont suivies une introduction au traitement d'image avec la bibliothèque OpenCV sur Python, la communication entre plusieurs systèmes par Bluetooth, la création d'une carte électronique sur EAGLE et de la modélisation SolidWorks pour accompagner le tout.

C'est de notre organisation, de nos échecs, de nos réussites et de nos conclusions quant à la découverte de ces nouvelles notions que nous parlons dans ce rapport.

B. Version Anglaise

It was our first actual project, and the challenges that we had to face were considerable. We are two students in our second year of preparatory school at Polytech Angers. In pairs, we haven't only created an Arduino tool to enable communication between several Dobot Magician robots, but we have also demonstrated the possibilities behind such a collaboration thanks to the building of a miniature city!

In this report, we come back on all the key moments of the past six months spent working on this project. The adoption of academic robots was followed by the discovery of C++ and Arduino programming. Starting from almost nothing, our ambition drove us to create an open-source library that enables controlling several robot instances from a single Arduino Board. These were followed by image filtering using the Python OpenCV library and communication among several devices through Bluetooth. We finally went through the creation process of an electronic board using EAGLE and some modelling using SolidWorks.

It is of our time and task management, of our failures, of our successes and of the conclusions we made about these new notions that we'll talk about in this report.

III. INTRODUCTION

A. Mise en contexte

L'intérêt, la recherche, la planification, la conception. Ce sont les notions que l'on nous enseigne lors de nos cinq années à Polytech Angers, et ce sont celles qu'il faut maîtriser pour intégrer sereinement le monde professionnel en tant qu'ingénieur. Chaque année, nous sommes introduits à de nouvelles expériences pour nous préparer aux attentes que l'avenir exigera de nous. Lors de la première, nous occupons un poste en entreprise pour découvrir un milieu professionnel, une organisation ou une équipe. En revanche, nous sommes maintenant en deuxième année, et c'est au semestre 4 que nous avons dû réaliser un projet dans son entièreté. Cette fois-ci, nous n'avions pas d'aide. Du début à la fin, il était notre mission de travailler en équipe, de rencontrer des problèmes, de chercher des solutions. Chacune des étapes de cette aventure fut passionnante, et c'est de celles-ci que nous allons parler dans ce rapport.

B. Choix d'un projet

Nous devions donc maintenant choisir un projet parmi une liste mise à disposition par les professeurs de Polytech Angers. Les possibilités sont nombreuses, les domaines sont différents, mais dans tous les cas, il faut faire preuve d'intérêt et de créativité pour réussir les objectifs proposés. En binôme, nous nous sommes concertés et avons décidé qu'il serait intéressant de découvrir ce que la robotique et l'électronique ont à nous offrir. C'est pour cette raison que, au milieu d'une multitude de propositions toutes plus intrigantes les unes que les autres, nous avons choisi pour notre projet, la Communication par Arduino entre plusieurs robots DOBOT Magician.

C. Nos attentes pour le projet

En effet, après avoir lu en détails la description de ce projet, plusieurs mots-clés nous sont restés en tête. Le premier est "robot", qui est exactement ce avec quoi nous espérions travailler lors de ces cent heures de projet. Le second est communication, qui implique les idées de collaboration et de synchronicité entre plusieurs robots, un concept attractif pour sa finalité tant visuelle que scolaire au niveau des connaissances qu'il pourrait nous apporter. Le dernier est évidemment Arduino, qui vient compléter les deux précédents. En effet, nous ne souhaitions pas être limités à utiliser des outils réalisés au préalable par d'autres personnes. Nous avions envie d'être impliqués, de créer quelque chose de toute pièce pour répondre aux objectifs fixés. C'était donc avec de grandes attentes que, parmi les 42 projets proposés, nous avons retenu celui-ci.

IV. DECOUVERTE DU PROJET

A. Explication des objectifs

Si nous souhaitions nous approprier le projet sur lequel nous allions travailler lors des six prochains mois, il fallait dès le début comprendre ses objectifs. Ces derniers, bien que suffisamment clairs et précis pour ne pas causer de mésentente, nous ont largement laissé de la place pour y incorporer une touche personnelle et des défis supplémentaires, que nous verrons en détails plus tard. De fait, l'objectif principal mentionnait la création d'un système de communication sur Arduino afin que plusieurs bras robotisés de la marque Dobot (Dobots Magician, ou Dobots) puissent accomplir une tâche en collaboration, qui serait autrement complexe si elle devait être réalisée par un unique Dobot. De cet énoncé, on peut déduire la nécessité d'un partage d'informations entre les robots, mais également la maîtrise de leurs mouvements respectifs, pour éviter l'interférence avec les actions de l'autre. Compte tenu de la tâche à réaliser, nous n'étions pas limités à ce sujet, tant que l'on parvenait à produire un résultat assez visuel pour potentiellement être présenté lors de futures portes ouvertes de l'école.

B. Présentation du matériel

La première étape lors de la réalisation d'un projet est d'appréhender autant d'informations que possible pour se familiariser rapidement avec le matériel que nous allons utiliser. D'une part, nous avions les bras robotisés produits par Dobot[1], comme mentionné précédemment. Il s'agit de petits robots articulés auxquels il est possible de fixer divers outils tels qu'une pince ou une ventouse afin qu'ils puissent saisir des objets, ou encore un crayon pour dessiner avec une répétabilité des mouvements de 0.2 mm, et une portée d'environ 31 cm.

Ce type de robot, les Dobots Magician, sont adaptés à une utilisation universitaire, et viennent donc avec un logiciel de contrôle (Dobot Studio), que nous avons utilisé en tant qu'outil de test au début de notre projet. D'autre part, nous avions à notre disposition une carte Arduino UNO, qui a été plus tard augmentée pour une carte Arduino Mega pour des raisons que nous verrons dans une section ultérieure. Elle occupe le rôle du "cerveau" de notre projet, en envoyant des instructions aux Dobots pour ensuite traiter leurs réponses et garder en mémoire le statut de complétion de la tâche à réaliser. Enfin, le dernier élément majeur auquel nous avons recours est une caméra, que nous utilisons dans le cadre d'une détection de formes dans l'espace grâce à un traitement d'image, ce qui joue un rôle essentiel dans le projet.

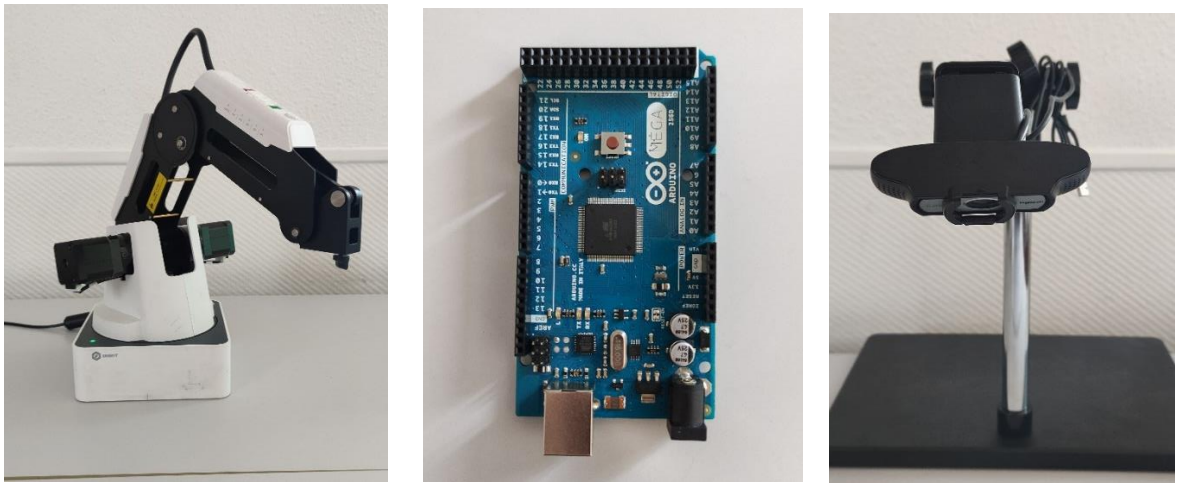


Figure 1 : Composants principaux de notre projet

C. Liste des enjeux

Comme nous l'avons vite compris, mener à bien ce projet relève de plusieurs enjeux importants dont le premier était qu'il puisse faire partie d'une démonstration lors de portes ouvertes à Polytech Angers. Il est donc nécessaire que la tâche réalisée en collaboration par les Dobots apporte une finalité visuelle, afin que le public puisse comprendre facilement de quoi il s'agit, mais également que ce qui entoure le projet soit propre et présentable. Cela inclut la connectique entre les Dobots et la carte Arduino, mais également une potentielle interface de contrôle, des messages d'information clairs et compréhensibles, etc. Il ne s'agit pas juste d'une finalité à atteindre, mais d'un travail pour rendre l'ensemble abordable par tous.

Un autre enjeu est la compréhension du fonctionnement des robots avec lesquels nous allons travailler, et la maîtrise de leurs mouvements. En effet, bien qu'il s'agisse de robots éducatifs, la documentation les concernant reste peu détaillée et l'un de nos plus gros défis reste finalement de s'assurer qu'ils puissent effectuer le travail demandé de manière régulière, répétable et sans imprécision majeure.

Enfin, en ajout à ceux que l'on vient de voir, nous avons décidé de nous rajouter un objectif supplémentaire, qui représentera un enjeu important tout au long de la conception du projet. Celui-ci est de créer une solution open-source pour le travail qui nous est demandé. Dans le monde actuel, une des plus belles avancées selon nous est la volonté de partager une solution dans le but de l'améliorer en communauté, et de la rendre accessible à tous. C'est pourquoi nous avons décidé de nous imposer une organisation modulaire concernant le code pour le projet, de sorte qu'il puisse être découpé en plusieurs bibliothèques réutilisables et une solution qui tire parti de ces dernières. Avec l'open-source viennent aussi des notions de documentation. Cela nous aura, certes, ajouté du travail, mais nous sommes maintenant fiers d'avoir publié nos premiers projets open-source, avec une documentation suffisamment détaillée pour que quiconque puisse les utiliser et se les approprier.

V. DETAIL DE LA METHODOLOGIE ABORDEE

A. Travail Préliminaire

i. Recherche d'un objectif pour les Dobots

Après avoir saisi dans son entièreté le projet, que ce soit au niveau de ses objectifs, des contraintes ou du matériel, il ne nous manquait plus qu'un élément pour pouvoir commencer à nous organiser concernant le travail à faire. Il s'agit de la tâche que les Dobots vont tenter de réaliser en collaboration. Nous avons cherché assidûment une solution adaptée et démonstrative du potentiel caché derrière le travail en équipe de deux simples robots éducatifs.

La première idée à laquelle nous avons pensé était de leur faire jouer une partie de *morpion*, ou chaque robot aurait tour à tour dessiné une croix ou un cercle pour indiquer son choix. Cependant, nous avons rapidement réalisé que la finalité de cette tâche n'était pas assez développée dans le cadre d'un projet de cent heures. Les notions requises étaient trop faibles et le résultat trop peu démonstratif pour qu'il soit d'un quelconque intérêt. D'autres idées que nous avons eues étaient par exemple le tracé de la solution d'un *labyrinthe en 2D*, la construction d'une petite *structure en bois*, ou encore l'approvisionnement en billes métalliques d'un *toboggan à billes*.

En revanche, un élément manquait à toutes ces idées : l'intervention humaine. Nous voulions que notre idée n'implique pas seulement deux robots autonomes dans leur tâche, mais que l'utilisateur puisse jouer un rôle dans la procédure s'il le souhaite. C'est ainsi que nous en sommes arrivés à notre idée finale, une idée assez ambitieuse qu'elle ne serait pas trop simple à créer, assez démonstrative pour répondre aux objectifs imposés et avec de nombreuses possibilités d'amélioration pour le futur. Cette idée est la construction d'une ville miniature grâce à des cubes en bois de plusieurs couleurs pour représenter différents types de bâtiments. L'un des robots s'occupera d'extraire d'une zone de stockage les blocs requis pour que le second construise la ville. La transition d'une zone à l'autre se fera grâce à un emplacement de transition auquel les deux Dobots auront accès. L'utilisateur peut alors, s'il le souhaite, interagir avec les robots en posant ou retirant des blocs dans la zone de stockage. Nous le verrons tout au long de ce rapport, mais cette idée présentait évidemment des défis que nous avons eus besoin de relever, ainsi que des possibilités d'amélioration à explorer.

ii. Prévision des besoins et des problématiques

Avec une idée de tâche à réaliser en tête, nous pouvions réfléchir aux éléments essentiels à sa réalisation. Même s'il est presque impossible de penser en amont à tous les besoins et aux points potentiellement problématiques, il est important d'essayer de se projeter dans le futur pour tenter d'anticiper les aspects qui seront à étudier plus en détails. En considérant cette approche proactive, nous avons identifié plusieurs éléments clés qui ont occupé une part majeure de notre travail lors de ce projet.

Premièrement, il nous aura fallu comprendre et adopter les protocoles de communication des robots. En effet, avec un robot comme lors d'interactions sociales, il est très compliqué de se comprendre si l'on ne connaît pas les moyens de communication de ses interlocuteurs.
Le second point essentiel que nous avons identifié est la perception de l'espace dans le référentiel du robot. Par la nature de notre projet, nous sommes constamment amenés à se faire déplacer les Dobots dans l'espace, il est donc important de pouvoir comprendre et maîtriser leur référentiel de mouvement.
Un troisième besoin, qui couplé à la vision par ordinateur pose un défi de taille, est la précision. Si nous souhaitons construire une ville miniature, il faut faire preuve d'une précision imparable pour ouvrir la porte à l'exécution d'actions avancées, comme l'empilement de cubes ou la rotation des bâtiments pour concevoir des structures complexes.

Enfin, un dernier élément sur lequel nous avons beaucoup travaillé est l'accessibilité du projet pour quiconque souhaite l'utiliser. Il en découle donc d'intégrer des interfaces utilisateur graphiques (GUI) pour un contrôle simple et visuel, ainsi que des indications compréhensibles tout au long de la procédure pour accompagner l'utilisateur en cas de questionnement.

iii. Mise en place d'une organisation

Bien que nous ne soyons que deux pour mener à bien ce projet, l'organisation et le tri des tâches à faire selon leur type et leur ordre de priorité est quelque chose que nous avons considéré essentiel depuis le début du projet. Aussi, nous avons créé un tableau de planification sur une application nommée Trello[2] :

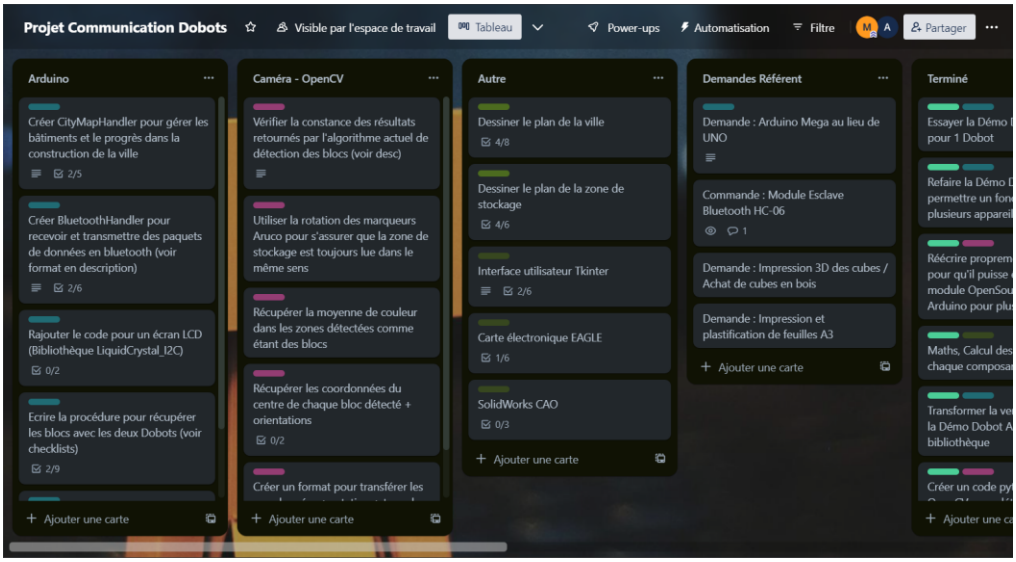


Figure 2 : Aperçu de notre organisation sur Trello

Chaque élément est rangé dans la liste qui lui convient le mieux, comme les listes Arduino, Caméra - OpenCV, ou encore Demandes Référent. De plus, il est possible d'intégrer à chacun de ces éléments une description, des labels et des check-lists :

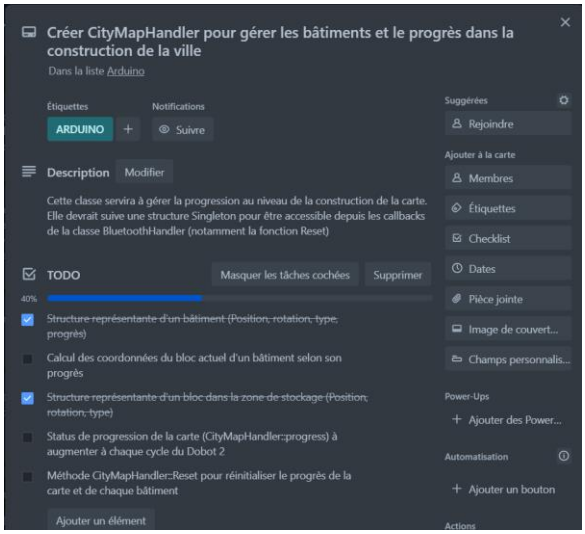


Figure 3 : Exemple du contenu détaillé d'un élément sur Trello

Grâce à la mise en place de cette organisation seulement une ou deux semaines après le début des projets, nous avons pu optimiser notre temps, prévoir à l'avance ce que nous devions faire lors des séances à venir, gérer nos priorités, etc.

B. Manipulation des robots Dobot Magician

i. Contrôle depuis l'interface Dobot Studio

Comme nous venons de le voir, le travail que nous avons réalisé en amont de notre projet était fastidieux, mais nécessaire au bon déroulement des mois à venir. Après avoir mûrement réfléchi aux besoins et aux multiples difficultés que nous allions rencontrer, il était temps de passer à la première étape dans la construction de notre ville miniature : se familiariser avec les robots. Cette étape était cruciale si nous souhaitions un jour être capable de manipuler avec précision plusieurs Dobots, et c'est avec **Dobot Studio** que nous avons commencé.

Figure 4 : Écran d'accueil de l'application Dobot Studio

Dobot Studio est une interface de contrôle qui offre plusieurs méthodes pour interagir avec un robot de la marque Dobot. Parmi ces méthodes, on retrouve notamment le **Teaching and Playback** (***Apprentissage et Répétition***), où le robot répète une série de mouvements préenregistrée, ou bien le **Write and Draw** (***Écrire et Dessiner***) qui laisse le robot dessiner un texte écrit ou une image téléversée dans l'application. Il y en a d'autres amusantes et éducatives, mais les deux qui nous intéressent le plus dans le cadre de ce projet sont **Blockly** et **Script**. Elles permettent chacune d'envoyer des instructions à un Dobot sous forme d'un programme rédigé dans le langage Python. Ce programme est en fait interprété par une bibliothèque qui transforme le code en paquets de données que le robot peut comprendre et exécuter en autonomie.

Pour nous aider à commencer, notre référent nous a fourni une carte électronique comportant divers boutons et LED dans le but de nous familiariser avec l'interface de contrôle des robots. Voici une image détaillant les entrées et sorties que l'on retrouve sur chaque Dobot Magician :

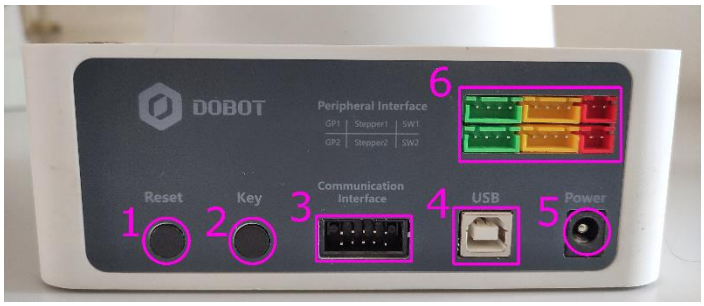


Figure 5 : Interfaces proposées par un Dobot

1. Bouton « Reset »
2. Bouton « Key »
3. Interface de communication
4. Port USB-B
5. Alimentation
6. Interface de contrôle des extensions

Bien que les usages puissent varier, chacune des zones encadrées ci-dessus est normalement réservée à certains types d'équipement. En zone 6 par exemple, se situe le nécessaire pour alimenter (*en rouge*) et contrôler (*en vert*) des extensions telles qu'une pompe à air ou un détecteur de couleurs. Les pins *jaunes* sont généralement utilisés pour coupler le robot à un rail linéaire dans le but d'augmenter sa portée latérale. Bien qu'il soit possible d'utiliser cette zone 6 à des fins de communication, la partie prévue à cet effet est celle numérotée 3 sur l'image précédente. Grâce à un câble de connexion à 10 pins, nous pouvons connecter le robot à la carte électronique très simplement de la manière suivante :

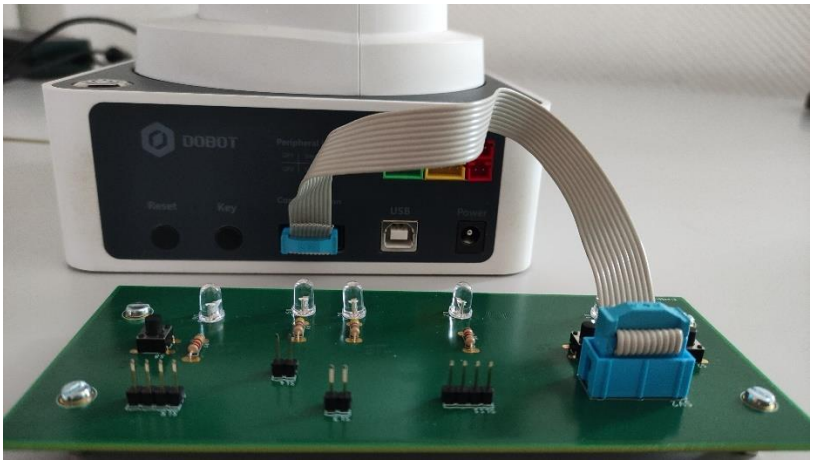


Figure 6 : Connexion d'un Dobot à la carte d'essais

Enfin, nous pouvons réaliser nos premiers tests avec l'interface de communication d'un Dobot, en faisant clignoter une LED à intervalles de 500 millisecondes avec le programme Blockly suivant :

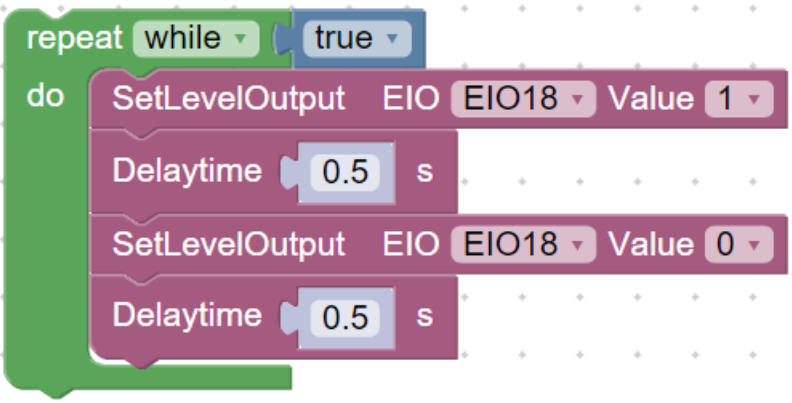


Figure 7 : Algorithme Blockly faisant clignoter la LED connectée à la sortie EIO18

C'est grâce à l'interface proposée par **Dobot Studio**, et au matériel mis à disposition par notre référent de projet, que nous avons pu observer la manière dont le robot se déplace, mettre en évidence son champ d'accessibilité, et plus généralement comprendre toutes les fonctionnalités des Dobots et les possibilités offertes par cette simple interface de communication de 10 pins.

ii. Contrôle depuis une carte Arduino MEGA

Dès lors que nous avions compris les capacités des Dobots que nous devions utiliser pour notre projet, ainsi que la méthode à mettre en place pour s'en servir au mieux, il était temps de laisser de côté l'interface **Dobot Studio**, et de s'aventurer en terres inconnues avec **Arduino**. Cette transition a été très difficile puisque nous n'avions aucune référence pour débuter et aucune connaissance des "normes" utilisées en robotique et en électronique pour faire communiquer entre eux deux appareils complètement différents. C'est ici que notre projet commença réellement, puisque c'est à ce moment que le travail de recherche et d'entreprise autonome démarra.

Après plusieurs heures à lire des documentations remplies de termes et de concepts inconnus, nous avons finalement fait la découverte d'un protocole de communication récurrent en robotique, le protocole **UART**[3] (**U**niversal **A**synchronous **R**eceiver / **T**ransmitter). Après plusieurs mois à travailler avec, cela nous paraît maintenant évident, mais ce protocole permet l'échange asynchrone de données en série entre deux dispositifs. Contrairement à certains protocoles qui sont cadencés par un signal d'horloge, une connexion UART est régie par un nombre d'informations par secondes (ou baudrate, en bits par seconde). Sur la *figure 5*, cet échange de données se fait via les pins **RX** et **TX** de la zone 3 du côté d'un Dobot, avec un baudrate de **115200 bits/s**. De plus, on retrouve des pins du même genre sur les cartes Arduino UNO et Arduino MEGA (voir figure ci-dessous). Nous l'aurons découvert plus tard, mais il est également possible de simuler d'autres interfaces de communication sur une carte Arduino (UNO ou Mega) grâce à une bibliothèque appelée "**SoftwareSerial**".

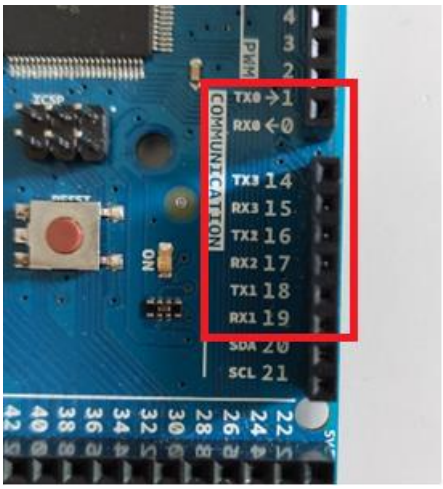


Figure 8 : Interfaces de communication UART sur une carte Arduino Mega

Avec cette découverte, nous étions sûrs que ces mystérieux pins étaient la clé pour résoudre notre premier problème. D'après les informations que nous avions trouvé en ligne, connecter les pins **RX** et **TX** du robot respectivement aux pins **TX** et **RX** d'une carte Arduino nous permettrait d'échanger des paquets de données.

En revanche, un problème persiste alors. *Comment pouvons-nous créer un système qui permet l'envoi facile de commandes à un robot via ce système de communication en série ?* Encore une fois, nous sommes retournés sur Internet. C'est alors que nous avons trouvé *un fichier très intéressant*[4], qui contient le format des

paquets de données compris par les robots ainsi que la liste complète des commandes pouvant être envoyées à un Dobot par ce biais. À peine avions-nous acquis une compréhension globale des normes de communication que nous sommes aussi entrés en possession d'un code de démonstration permettant de faire bouger un Dobot en le connectant à une carte Arduino Mega. Le contrôle depuis une carte Arduino UNO est possible, mais la RAM mise à disposition par ce modèle est trop limitée pour permettre la connexion de plusieurs Dobots. En plus d'une démonstration, ce code contient également un début d'intégration d'un certain nombre de commandes pour le robot, tel que l'envoi des paramètres de vitesse de déplacement, la rotation des joints ou encore le déplacement vers un point dans le repère du robot. Cette trouvaille nous a non seulement permis de ne pas commencer de zéro, mais surtout de nous fournir une base solide sur laquelle se reposer lors des mois à venir pour contrôler nos robots.

Nous avons passé plusieurs semaines à lire et relire ligne par ligne le code de démonstration dans l'objectif de comprendre l'utilité de chaque morceau de code en imaginant le procédé global. Petit à petit, nous avons tenté de faire des changements pour mieux comprendre certains choix de design fait par les développeurs au niveau du code. Premièrement, nous avons retiré une boucle infinie dans la fonction `loop()` du programme. Cette dernière s'exécute indéfiniment et de manière répétée après l'initialisation de la carte Arduino. Une boucle infinie dans cette fonction n'a alors aucune utilité, et serait même gênante pour un code plus complexe que celui proposé par la démonstration.

Nous avons ensuite étudié en détail le code se chargeant d'envoyer des commandes au Dobot connecté. Après plusieurs essais et l'introduction d'instructions pour déboguer la procédure, nous nous sommes rendu compte que les réponses envoyées par le Dobot lorsqu'un paquet lui est transmis étaient lues uniquement quand de nouvelles instructions lui sont envoyées. Ce problème n'était pas important dans le cadre du code de démonstration. Cependant, si dans le futur nous avions eu besoin de récupérer des informations concernant le robot, comme sa position dans l'espace ou la rotation de ses joints, il aurait été très probable que la réponse du robot soit tout simplement perdue puisqu'elle ne serait pas lue immédiatement. Une simple correction pour ce problème a été d'ajouter un morceau de code dans une fonction qui s'exécute toutes les 100 millisecondes pour lire les données provenant du robot sur l'interface UART utilisée.

Enfin, nous avons compris le système de Ring Buffer[5] (*Tampon Circulaire*) utilisé par la démonstration pour stocker et envoyer les paquets de données au Dobot. Il ne s'agissait finalement que d'un choix d'infrastructure fait par les développeurs, que nous avons décidé de garder par la suite pour des raisons de simplicité, puisque le code était déjà prêt et réutilisable.

Encore une fois, nous avons fait le choix de passer beaucoup de temps sur la compréhension de ce code de démonstration. Ce travail fastidieux nous a permis de nous l'approprier en déterminant l'importance de chaque morceau de code et son impact à l'échelle de l'infrastructure globale du programme. C'est grâce à cela que nous avons été capables de le modeler en quelque chose qui convenait à nos attentes et à nos besoins.

iii. Création de la bibliothèque DobotNet pour Arduino

Avec tout ce temps passé à étudier le même programme Arduino, il n'avait plus aucun secret pour nous. En revanche, il ne faut pas perdre de vue que l'objectif principal de notre projet est de contrôler plusieurs Dobots en les faisant communiquer grâce à une carte Arduino. C'est à ce moment que nous avons eu l'idée de créer une bibliothèque qui offrirait ce qu'aucune ne permettait jusqu'à présent, c'est-à-dire une **API** (**A**pplication **P**rogramming **I**nterface) open-source pour gérer plusieurs Dobots Magician depuis une unique carte Arduino Mega, à l'aide d'une structure simple à utiliser, de fonctions documentées et d'exemples pour aider à la prise en main. C'est ainsi que nous avons commencé à planifier, en commentant dans la démonstration toutes les parties qui pourraient être utilisées sans changement, celles qui nécessiteraient une adaptation et enfin celles

qui seraient à réécrire entièrement. Voici les étapes que nous avons suivi pour écrire la bibliothèque DobotNet :

Dans un premier temps, décrivons ce qui caractérise l'instance d'une classe représentant un Dobot connecté à la carte Arduino. La seule donnée dont on a réellement besoin est *l'interface UART* à laquelle il est connecté pour savoir où transmettre les données. Si l'on souhaite contrôler plusieurs robots, il peut également être utile de leur attribuer *un identifiant*. Enfin, on peut souhaiter avoir plus de Dobots que d'interfaces UART présentes par défaut sur la carte, ajouter un support pour des interfaces émulées grâce à la classe *SoftwareSerial* serait donc un plus.
Ainsi, indiquer à la bibliothèque qu'un robot est connecté aux pins TX1 et RX1 de la carte Arduino (Montage Annexe 1) devrait être aussi simple que d'écrire les deux lignes suivantes :

```
HardwareSerialWrapper S1{&Serial1}; // Indiquer qu'il s'agit d'une interface
UART existante
DobotInstance dobot{&S1, 0}; // Créer l'instance représentant le Dobot
d'identifiant 0 connecté au Serial1
```

De plus, il serait très pratique s'il était possible d'envoyer tout type de commande au robot en appelant une simple fonction associée à la classe du Dobot. Par exemple, si l'on souhaite faire bouger le robot vers un point x, y, z avec une rotation de l'outil de 0°, en utilisant la méthode JUMP, la syntaxe devrait être la suivante :

```
dobot.MoveTo(JUMP_XYZ, x, y, z, 0);
```

En outre, si l'on souhaite récupérer les coordonnées de la tête du Dobot et la rotation de ses joints, la syntaxe pourrait être la suivante, où "*false*" indique si l'on doit attendre que le robot ait terminé les actions envoyées précédemment ou non.

```
Dobot.SendGetterProtocol(ProtocolGetPose, false);
```

Enfin, il devrait y avoir deux possibilités pour confirmer la transmission des paquets de données vers les robots. La première serait l'envoi individuel, et la seconde l'envoi pour tous les Dobots enregistrés sur le réseau géré par la bibliothèque. Pour être vraiment complet, ce serait super que l'on puisse passer en paramètre la référence à une fonction qui servirait de callback (fonction exécutée après la complétion du code) pour toutes les données envoyées par un Dobot et reçues par l'Arduino. Idéalement, cela devrait ressembler à ce code :

```
// Déclaration du squelette de la fonction callback
void Handle_DobotRx(uint8_t dobotId, Message* msg);
DobotResponseCallback callback = Handle_DobotRx;

// Envoi individuel
dobot.SerialProcess(&callback);

// Envoi groupé
DobotNet::Tick(&callback);
```

Tandis que certaines personnes ne sauraient pas par où commencer pour obtenir un résultat comme celui présenté précédemment, nous avons la chance d'avoir dans notre binôme une expérience déjà avancée en développement informatique. Bien qu'il s'agisse de notre premier projet où le code est dans le langage C++, les méthodes de pensée et l'organisation du code restent similaires aux autres langages de programmation.

Ainsi, nous avons commencé par créer une classe **DobotInstance** dont, comme expliqué précédemment, chaque instance représente un Dobot connecté à une interface UART (*matérielle ou émulée*) de la carte Arduino. Nous avons ensuite modifié le code de traitement des paquets de données de sorte à pouvoir transmettre des commandes sur n'importe quelle connexion UART spécifiée. Enfin, nous avons réécrit directement dans la classe **DobotInstance** les fonctions permettant l'envoi facile, rapide et automatisé de paquets de données correspondant aux diverses actions réalisables par un Dobot. D'autres changements et optimisations ont été réalisés, mais nous ne pouvons pas les lister puisqu'ils sont nombreux et sans intérêt majeur dans le cadre de ce rapport.

Nous sommes heureux d'avoir accompli notre premier objectif, qui est la création d'une bibliothèque pour contrôler un ou plusieurs Dobots depuis une unique carte Arduino Mega. Il s'agit de **DobotNet**[6], et elle est disponible sur *GitHub*. Nous avons également pris le temps de la documenter et d'écrire des exemples de programme, allant de la réplication du code de démonstration en quelques lignes seulement (**Annexe 2 et 3**), jusqu'au code complet de notre projet final. Évidemment, nous n'avons abordé que la création de la bibliothèque jusqu'à maintenant, et nous verrons dans les parties suivantes notre progression pour créer la solution à notre problème, qui est la construction d'une ville miniature par deux Dobots.

C. Vision par ordinateur et traitement d'image

i. Origine du besoin

Comme nous l'avons expliqué au début de ce rapport, notre idée est de construire une ville miniature grâce à la collaboration de deux Dobots, où chaque type de structure dans la ville est représenté par une couleur.

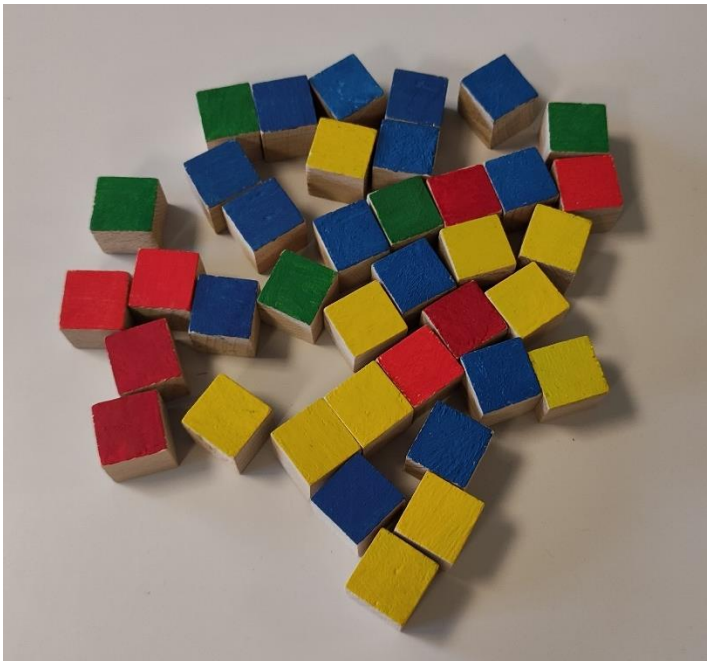


Figure 9 : Aperçu des blocs de couleur utilisés pour construire notre ville miniature

Pour ce faire, tandis que l'un des robots s'occupera de placer les blocs dans la ville, l'objectif de l'autre est d'assurer son approvisionnement en blocs de construction via un emplacement commun, la *zone de transition*. En revanche, les blocs placés dans ce que l'on appelle la *zone de stockage* n'ont pas de position ou de couleur prédéfinie puisque l'utilisateur peut les placer où il le souhaite. Ainsi, arrive un nouveau besoin : déterminer la position, la rotation et le type des blocs placés dans la zone de stockage.

Plusieurs solutions se présentent à nous pour répondre à ce problème, mais celle que nous avons retenu est la **vision par ordinateur**[7], à l'aide d'une caméra placée au-dessus de cette zone.

En effectuant quelques recherches sur le sujet, nous avons découvert qu'un outil parfois utilisé en vision par ordinateur pour délimiter une zone ou repérer un objet mobile sont les marqueurs Aruco. Il s'agit de petits carrés en 2D dont la taille est généralement connue et qui, semblables à un QR Code, transmettent grâce à leur arrangement plusieurs informations, comme le numéro qui leur correspond, leur sens, leur orientation dans l'espace, etc. Voici des exemples de marqueurs Aruco issus du dictionnaire 4x4 :

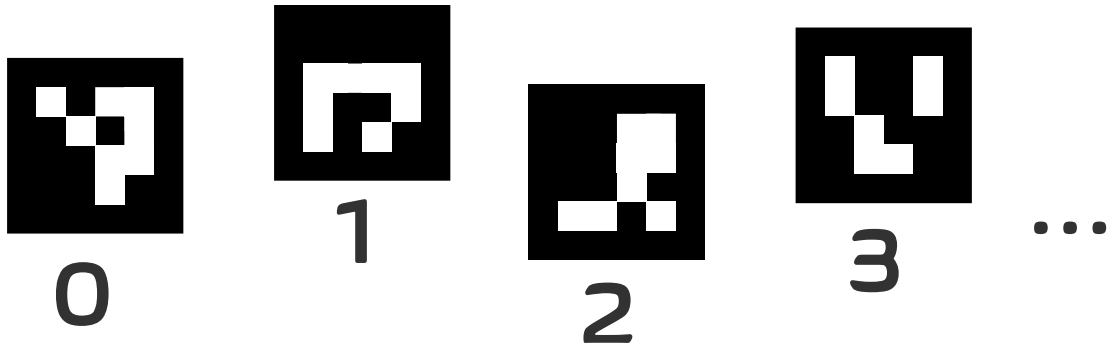


Figure 10 : Exemple de marqueurs Aruco, associés à leur valeur numérique

Nous avons fait de manière prématurée le choix de délimiter notre zone de stockage grâce à ces marqueurs, de sorte à s'assurer que le robot ne puisse pas aller chercher des blocs en dehors de son champ d'accessibilité. Nous sommes maintenant conscients que nous n'avons pas utilisé tout le potentiel offert par ces marqueurs (dimensions, repères associés, …). De toute manière, voici une image illustrant la situation :

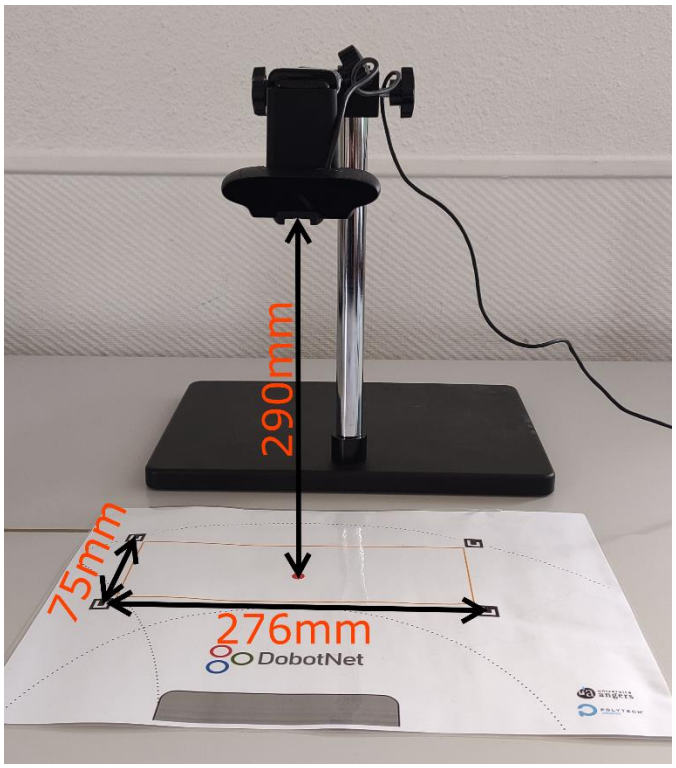


Figure 11 : Représentation des dimensions connues lors d'une capture depuis la caméra

ii. Introduction à la bibliothèque OpenCV sur Python

Maintenant que nous avons délimité la zone de stockage, nous pouvons nous aventurer à traiter le flux d'images fourni par la caméra de sorte à détecter tout ce qui ressemble à un cube vu du dessus. Cette opération a été possible grâce à la bibliothèque de traitement d'image OpenCV (Open Computer Vision), disponible notamment sur Python. Elle met à disposition de nombreux algorithmes préprogrammés qui permettent d'effectuer des opérations en tout genre sur une image.

Évidemment, au début de ce projet, nous n'avions absolument aucune connaissance en traitement d'image, et par conséquent aucune expérience avec OpenCV. Nous avons donc commencé en copiant des morceaux de code venant d'Internet, en espérant que par miracle tout se mette à fonctionner. Il va de soi que ce n'était pas la bonne solution, puisqu'après quelques séances à travailler dessus, nous n'avions aucun résultat prometteur. Par la suite, nous sommes allés rechercher des informations et apprendre des concepts du domaine de la vision par ordinateur, en recommençant toute cette partie de zéro et en laissant de côté notre manque de motivation.

La première étape vers la réussite de cette partie de notre projet est la détection et la découpe de la zone de stockage sur le flux d'images pris par la caméra. Heureusement pour nous, la bibliothèque OpenCV propose une section réservée à la détection et au traitement des marqueurs Aruco. Ainsi, il nous a été facile de découvrir la fonction `cv2.aruco.detectMarkers()`, qui, comme son nom l'indique, permet de détecter des marqueurs Aruco dans une image. Elle retourne les coins de chacun des marqueurs trouvés, ainsi que la valeur portée par chacun d'entre eux. Il est donc facile de découper la capture de la caméra de sorte que les quatre marqueurs entourant la zone de stockage correspondent aux extrémités de la nouvelle image. Malheureusement, c'est sur cette étape que nous avons rencontré de nombreux problèmes, et nous allons essayer de voir les principaux dans les paragraphes suivants.

Au début, nous avions choisi de découper la zone en traçant le rectangle ayant la plus petite aire qui entourait le centre des quatre marqueurs détectés. Néanmoins, à cause de la projection faite par la caméra sur un plan 2D de la scène en 3D, la zone de stockage au niveau des pixels ne semblait pas être un rectangle, mais plutôt un trapèze. Ainsi, la découpe "linéaire" que nous faisions n'était pas précise, donnant un résultat similaire à celui mis en évidence ci-dessous :

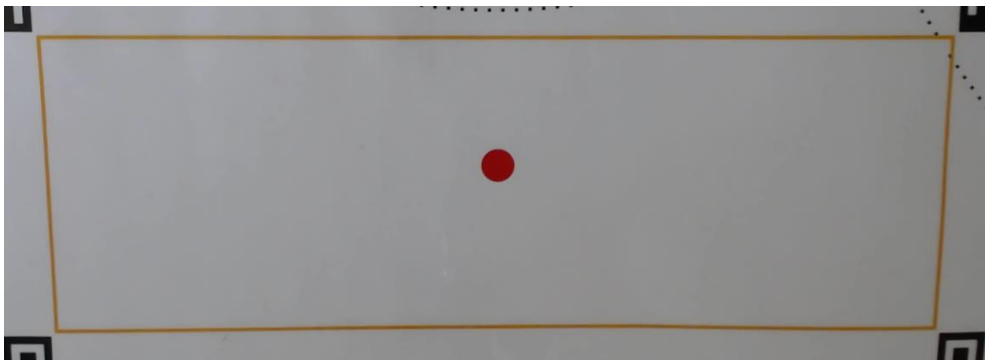


Figure 12 : Découpe linéaire de la zone de stockage depuis une capture de la caméra

Les dimensions étaient donc fausses, ce qui aurait mené à de nombreux problèmes pour le futur, que ce soit au niveau de la détection des blocs de construction ou la détermination de leur position dans l'espace.

Il fallait donc trouver une solution pour découper la zone de stockage de sorte à conserver au mieux ses dimensions et celles de son contenu. Comme nous venons de le voir, la zone capturée par la caméra a plutôt

une forme trapézoïdale que rectangulaire. Une tentative naïve a donc été de simplement utiliser la fonction `cv2.warpPerspective(image, M, (width, height))`, qui, combinée à `M = cv2.getPerspectiveTransform(source, dest)`, permet d'appliquer une matrice de transformation sur l'image telle que des points sources (ici, le centre de nos marqueurs Aruco) soient déplacés vers une destination donnée (ici, les coins de notre image). Cependant, voici le résultat chaotique que nous obtenions avec cette méthode :

Figure 13 : Transformation d'une capture de la zone de stockage sans ordonner les marqueurs détectés

Effectivement, jusqu'à présent, nous demandions à OpenCV de déplacer le centre de nos marqueurs vers les coins de notre image sans se préoccuper de l'ordre dans lequel les marqueurs étaient détectés. Tout ce que nous demandions, c'était de déplacer le premier marqueur en haut à gauche, le second en haut à droite, le troisième en bas à droite, et le dernier en bas à gauche de l'image. Cependant, si la caméra était bien placée, les marqueurs étaient retournés dans l'ordre suivant : le premier en haut à gauche, le second en haut à droite, le troisième en bas à gauche, le dernier en bas à droite. De plus, le résultat était d'autant plus imprévisible que la caméra ne peut pas être placée à chaque fois au même endroit avec la même orientation, ce qui provoque une détection des marqueurs dans un ordre plus ou moins aléatoire.

Après quelques moments de réflexions, nous avons enfin trouvé l'élément manquant : ordonner les centres des marqueurs dans le sens horaire. Nous avons résolu ce besoin grâce à un simple algorithme trouvé sur Internet qui trie une liste de points passée en paramètre (de la forme `[[x1, y1], [x2, y2], ...]`). La source et sa transcription commentée sont disponibles en **Annexe 4**. Après avoir appliqué cet algorithme, nous avons finalement réussi à obtenir le résultat attendu :

Figure 14 : Transformation d'une capture de la zone de stockage après avoir ordonné les marqueurs Aruco

Désormais, nous avons accès à la zone de stockage isolée et redressée. Il ne nous reste alors plus qu'à trouver un moyen de détecter les cubes qui se situent à l'intérieur de celle-ci. Nous allons utiliser la scène suivante pour illustrer les propos des prochains paragraphes :

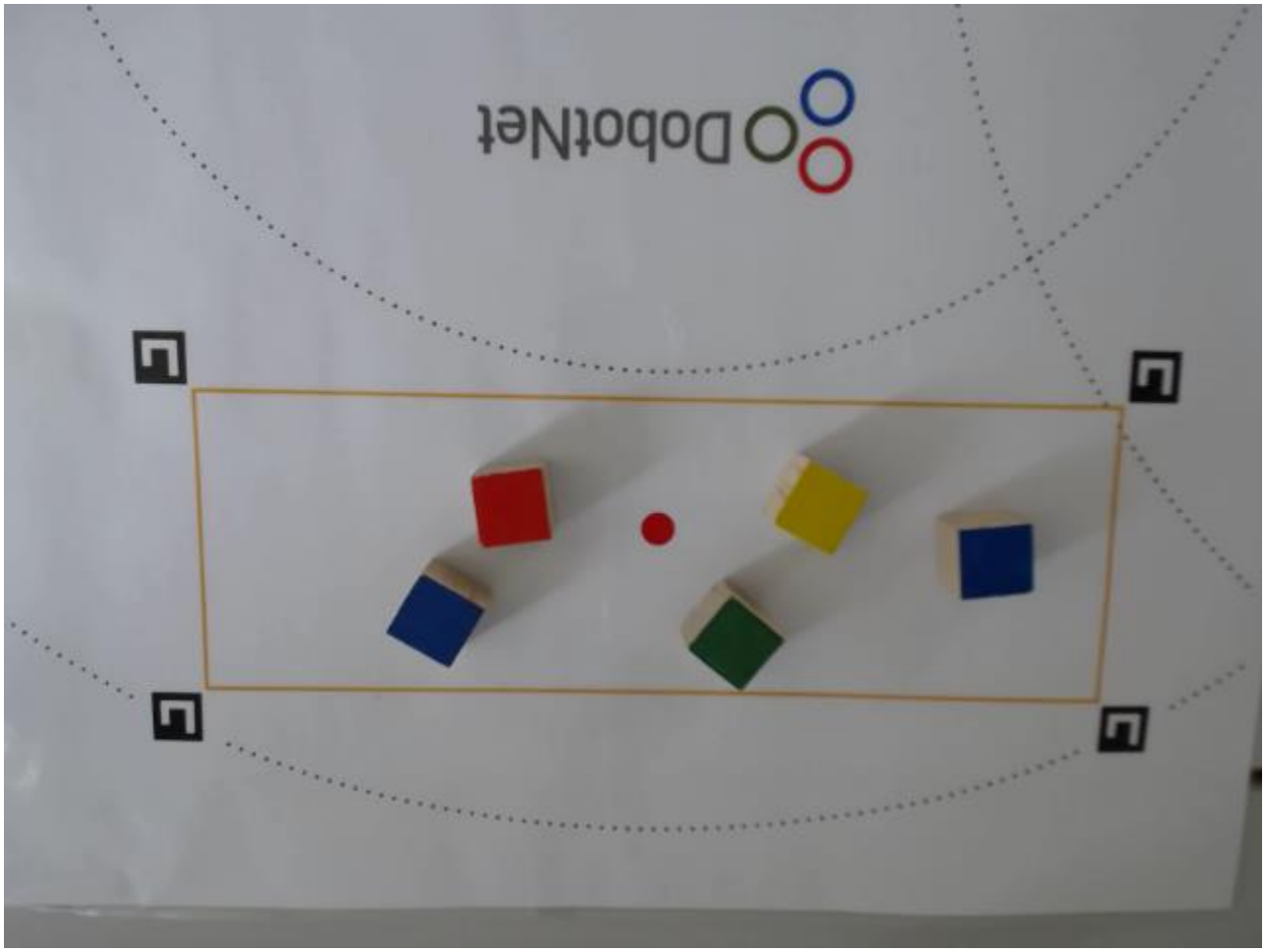


Figure 15 : Capture d'une scène usuelle par la caméra

La détection de formes de référence dans une image se fait généralement avec la fonction `cv2.findContours()`, qui cherche dans une image donnée tout changement soudain de couleur pouvant s'apparenter au contour d'un objet. Utilisée seule, cette fonction est peu efficace dans des conditions réelles puisque la caméra ne peut pas capturer de photos parfaites. Toutes les petites variations aléatoires capturées par la caméra (ou *bruit*) vont causer des résultats peu intéressants puisqu'ils seront complètement dépendants d'innombrables facteurs externes, tels que les *conditions d'éclairage*, *l'orientation* de la caméra, etc. Pour illustrer ce propos, nous pouvons regarder l'image ci-dessous, qui entoure en vert tous les contours détectés.

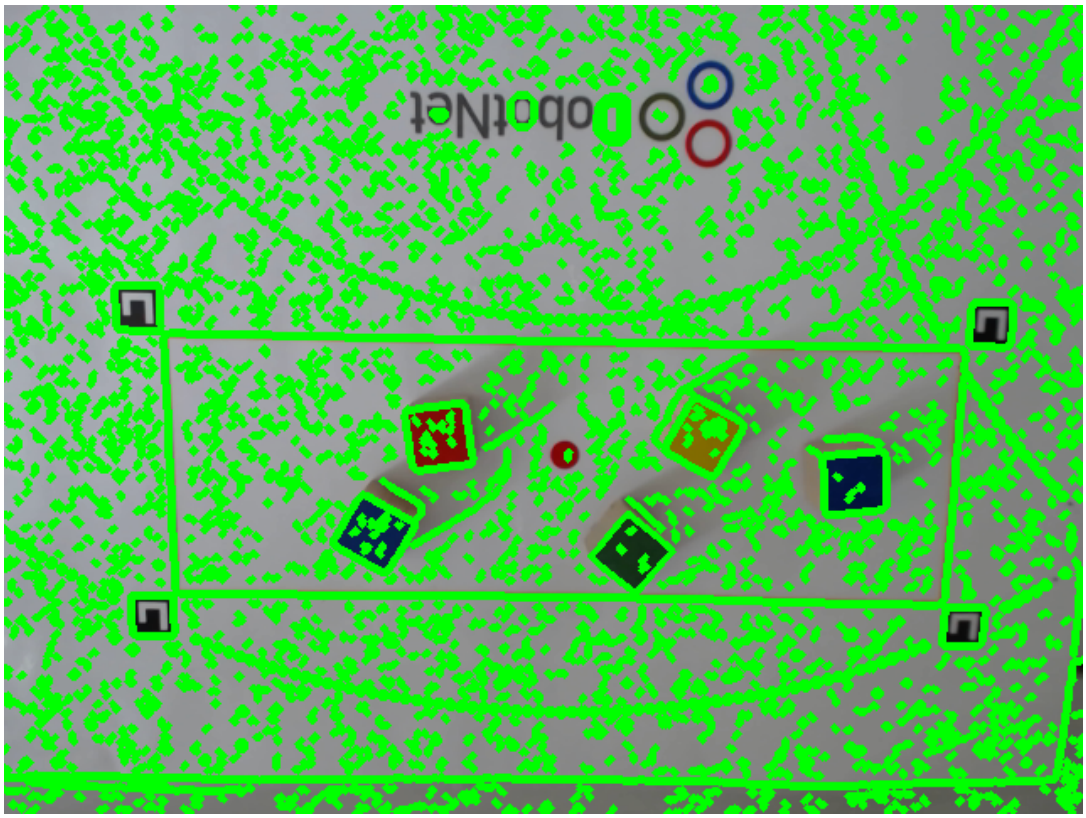


Figure 16 : Affichage des contours détectés sur la scène

Il nous faut donc premièrement un moyen pour réduire le bruit présent sur l'image. Comme nous l'avons découvert, cela peut être réalisé en appliquant un filtre pour légèrement flouter l'image. La technique que nous avons décidé d'utiliser est le flou gaussien (ou **Gaussian Smoothing**[8] en anglais). Cela revient à appliquer une fonction sur chacun des pixels de l'image de sorte qu'il prenne la valeur de la moyenne des pixels adjacents dans un rayon défini. Cette méthode permet *d'atténuer le bruit* présent sur une image et, réciproquement, de mettre en évidence les contours des blocs que l'on veut détecter. Voici notre scène une fois qu'un filtre gaussien a été appliqué :

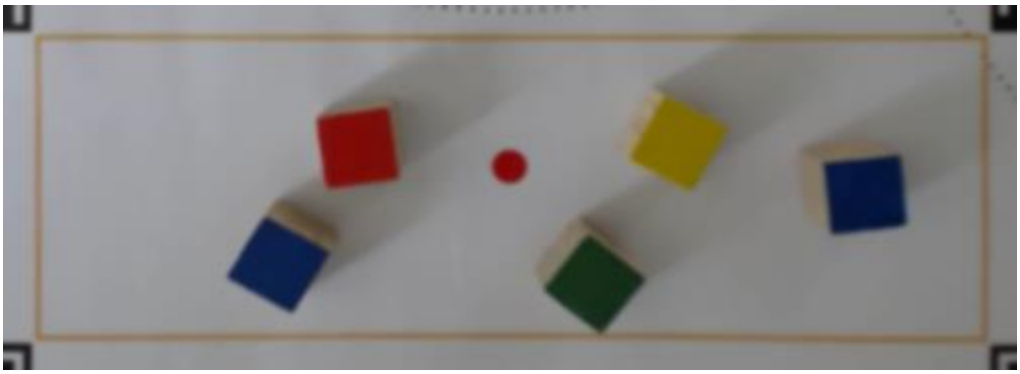


Figure 17 : Résultat après application d'un flou gaussien à la scène

Maintenant que le bruit sur notre image a été bien atténué par le flou gaussien, nous pouvons utiliser un algorithme de détection des contours. OpenCV en propose plusieurs, mais après quelques essais (entre la détection des contours directe, la méthode de Sobel et la **méthode de Canny**[9]), nous avons déterminé que

l'algorithme qui nous apportait les meilleurs résultats était celui de Canny. En appliquant la fonction `cv2.Canny()` à notre image floutée, ceci est un des résultats que nous pouvons obtenir :

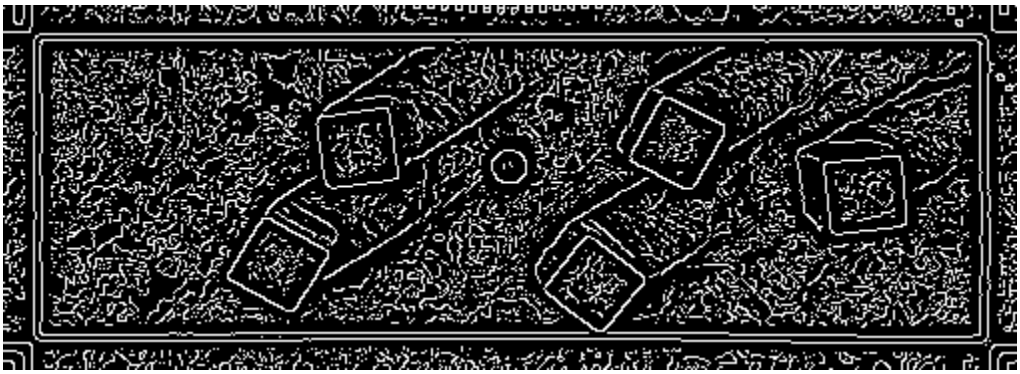


Figure 18 : Application d'un algorithme de détection des contours (Canny) à la scène

Pour fonctionner, l'algorithme de Canny utilise deux valeurs considérées comme bornes supérieures et inférieures. Si le gradient d'un pixel[10] (=*changement directionnel d'intensité*) est plus élevé que la borne supérieure, alors il est accepté comme faisant partie d'un contour. En outre, s'il est plus bas que la limite inférieure, alors il est rejeté. Puisque le bénéfice majeur de cette fonction était de supprimer les ombres causées par la hauteur des blocs, nous avons décidé plus tard de référer à ces paramètres comme étant la "*taille des ombres*" et leur "*intensité*". Même si ces définitions ne sont pas exactes, elles sont assez représentatives de leur impact sur l'image dans notre cas.

Après un ajustement de ces deux paramètres, la fonction `cv2.Canny()` retourne un résultat similaire au suivant :

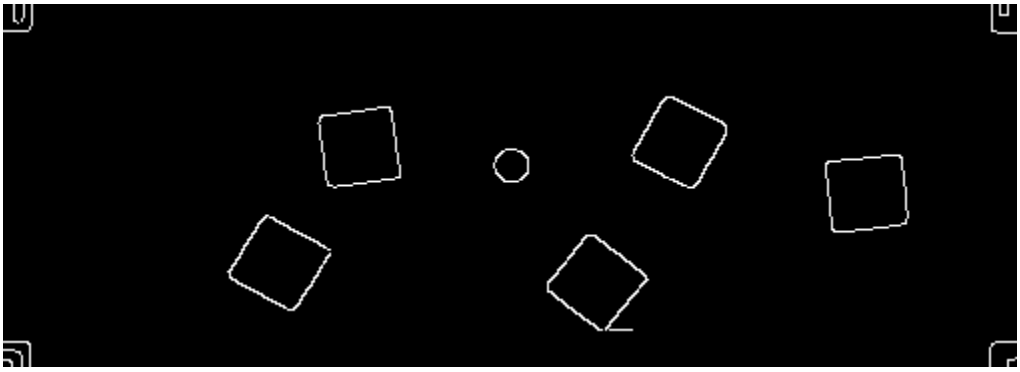


Figure 19 : Application de l'algorithme de détection des contours avec un réglage adapté à l'environnement

Puisque nous ne sommes pas expérimentés en vision par ordinateur, il est possible que des algorithmes plus efficaces pour notre situation existent, ou que d'autres auraient pu être ajoutés en complément pour améliorer les résultats. En revanche, nous avons décidé de conserver cette technique qui produit des réponses assez proches de la réalité dans le cadre de notre projet.

iii. Extraction des informations relative aux cubes

Avec l'ajout des traitements abordés précédemment, les contours retournés par la fonction `cv2.findContours()` sont maintenant bien plus fiables. Pour les valider une dernière fois et s'assurer qu'il s'agisse bien de blocs présents dans la zone de stockage, nous avons ajouté trois conditions majeures.

La première a pour but de vérifier que le contour a entre quatre et neuf côtés, qui est le nombre maximum de côtés visibles lors de la projection d'un cube sur un plan 2D.

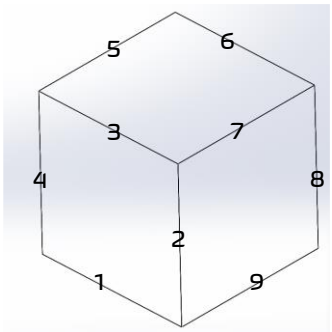


Figure 20 : Numérotation des côtés visibles d'un cube projeté en 2D

La seconde vérifie que l'aire du plus petit carré contenant tous les côtés du contour (1) est proche de celle attendue, c'est-à-dire d'environ 400 mm² pour des cubes de 20 mm de côté. Cette vérification est possible puisque nous connaissons les dimensions entre chaque marqueur Aruco délimitant la zone de stockage, ainsi que le nombre de pixels distant de chacun des centres des marqueurs. Ces mesures nous laissent établir un ratio pour passer d'une mesure en pixel à une mesure en millimètres, et inversement :

```
# Dimensions de la zone de stockage, en mm
storage_dimensions = 278, 104
# Calcul du ratio horizontal et vertical (width et height sont les dimensions de l'image transformée, en pixels)
ratio = storage_dimensions[0] / width, storage_dimensions[1] / height
```

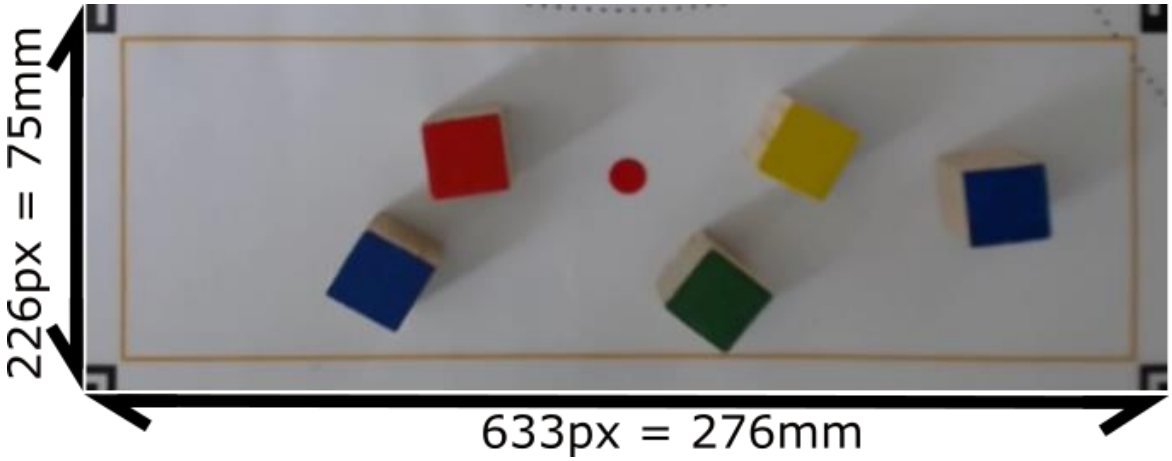


Figure 21 : Mise en évidence des ratios de conversion (pixels -> millimètres)

Enfin, la dernière vérification s'effectue sur le rapport entre la hauteur et la largeur du contour détecté. Si celui-ci s'éloigne trop de celui d'un carré (plus de 35 % de différence), alors le contour est rejeté.

Avec les contours détectés et validés, la partie la plus compliquée du travail est déjà faite. Grâce à (1), le résultat obtenu avec la fonction `box = cv2.minAreaRect(contour)`, nous pouvons récupérer simplement les coordonnées (cX, cY) du centre du cube sur la projection 2D, ainsi que sa rotation rot, avec la ligne suivante :

```
((cX, cY), (_, _), rot) = box
```

La dernière information dont nous avons besoin désormais est le type du bloc déterminé par sa couleur. Pour ce faire, nous avons calculé la couleur moyenne contenue dans la zone où se situe le bloc en appliquant un masque à l'image autour du contour. Le code pour cette action est assez simple :

```
# Obtenir la couleur moyenne du contour en créant un masque autour
rect = cv2.boxPoints(box).astype(np.int0)
# Remplir le masque en noir
mask = np.zeros_like(image)
# Remplir le masque en blanc où se situe le contour
cv2.fillPoly(mask, [rect], (255, 255, 255))
# Transformer les couleurs du masque en variances de gris
mask = cv2.cvtColor(mask, cv2.COLOR_BGR2GRAY)
# Récupérer la couleur moyenne de l'image avec le masque spécifié
mean_color = cv2.mean(image, mask=mask)
boxType = type_from_color(mean_color)
```

Où type_from_color est une fonction regardant simplement quel type de bâtiment est attribué à la couleur la plus proche de mean_color. Par exemple, dans un espace à 3 dimensions BGR (Bleu Vert Rouge), le type maison est attribué à la couleur (0, 0, 255). Si l'on détecte une couleur moyenne dans le repère BGR de (16, 29, 120) sa distance aux quatre types de bâtiments est la suivante :

$$dist_{red} = \sqrt{16^2 + 29^2 + (255 - 120)^2} = 139.0$$
$$dist_{green} = \sqrt{16^2 + (255 - 29)^2 + 120^2} = 256.4$$
$$dist_{yellow} = \sqrt{16^2 + (255 - 29)^2 + (255 - 120)^2} = 263.7$$
$$dist_{blue} = \sqrt{(255 - 16)^2 + 29^2 + 120^2} = 269$$

On en déduit donc que la couleur détectée est plus proche du rouge que des autres types, le bloc est alors certainement un morceau de maison. Nous pouvons enfin afficher toutes ces informations sur le flux d'image de la caméra à des fins de visualisation, et obtenir un résultat comme le suivant :

Figure 22 : Affichage des résultats obtenus après l'application des filtres abordés

D. Communication et intégration des composants

Maintenant que nous disposons de toutes les informations nécessaires concernant la zone de stockage sur l'ordinateur relié à la caméra, il nous faut trouver un moyen pour les communiquer à la carte Arduino. Dans le but de réduire le nombre de connexions filaires ainsi que pour expérimenter avec des modules Arduino, nous avons fait le choix de relier ces deux éléments majeurs de notre projet par Bluetooth.

Pour ce faire, nous nous sommes procuré grâce à notre référent de projet un module Bluetooth esclave pour Arduino (HC-06). Comme pour les Dobots, le module HC-06 utilise le protocole UART pour communiquer en série les données reçues par Bluetooth avec la carte Arduino. Aussi, en émulant une interface de communication UART sur les pins 2 et 3, voici à quoi ressemblerait le branchement entre les deux appareils :

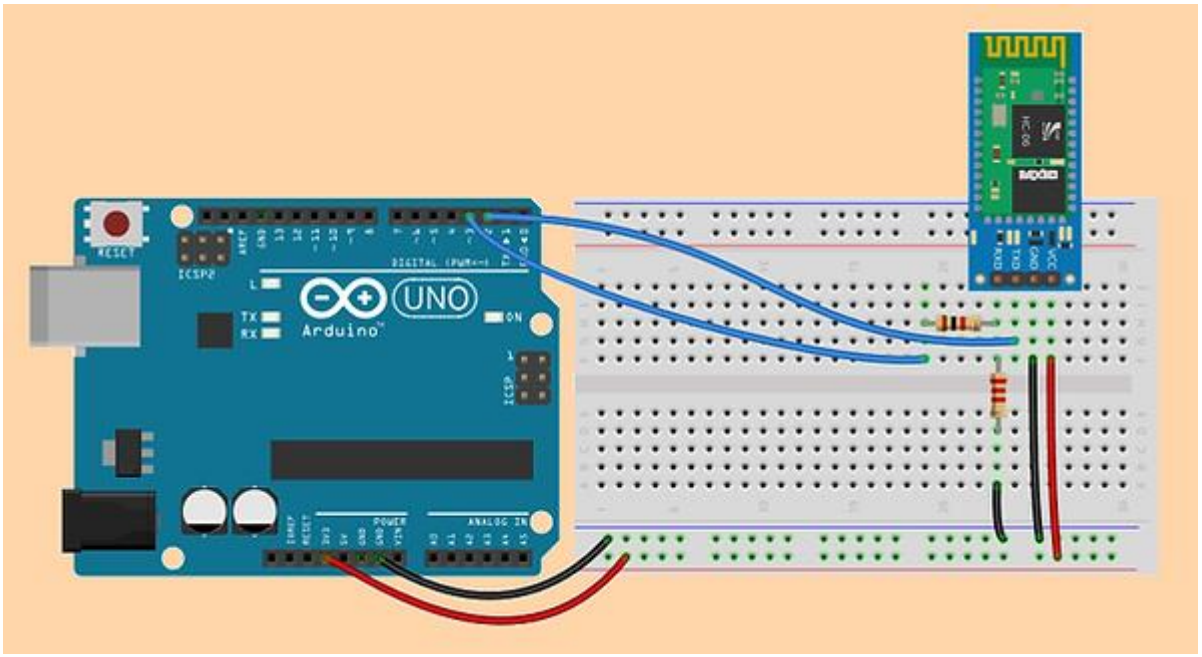


Figure 23 : Branchement entre le module Bluetooth HC-06 et une carte Arduino UNO

L'arrangement des résistances sur le schéma ci-dessus permettent d'atteindre une tension entrante dans le pin RX du module HC-06 de 3.3V, puisqu'une tension de 5V serait trop élevée.

Un module pour communiquer en Bluetooth n'était pas notre seul besoin pour établir un flux de données entre l'Arduino et le PC. En effet, comme nous l'avons appris avec le protocole de communication en série des Dobots, il faut organiser les données si l'on souhaite avoir des échanges propres et compréhensibles. Pour cette raison, nous avons créé notre propre format de paquet de données pour cette communication Bluetooth. Il reste évidemment très simple en vue du peu de données que nous aurons besoin de transmettre, mais c'était une fonctionnalité intéressante à implémenter des deux côtés du réseau.

Notre format s'établit de la manière suivante :

- Chaque donnée est écrite en base hexadécimale
- Chaque paquet de données commence par un "**header**" 0xAA et se termine par un "**footer**" 0xAA
- Le premier caractère après le "header" permet d'identifier le format du paquet. Par exemple, un paquet commençant par "AA**E**" aura l'identifiant **14**. Cet identifiant permet de repérer la taille espérée pour ce paquet ainsi que la fonction qui se chargera de le gérer une fois reçu entièrement
- Les caractères qui suivent sont les **paramètres** du paquet. Leur ordre et taille sont prédéfinis, et ils se lisent de gauche à droite (Big-endian)
- Le dernier caractère avant le "**footer**" correspond à la somme de contrôle (**checksum**). Elle se calcule simplement en additionnant les bits individuels de l'équivalent numérique pour chacun des caractères contenus dans le **payload (identifiant + paramètres)** du paquet de données

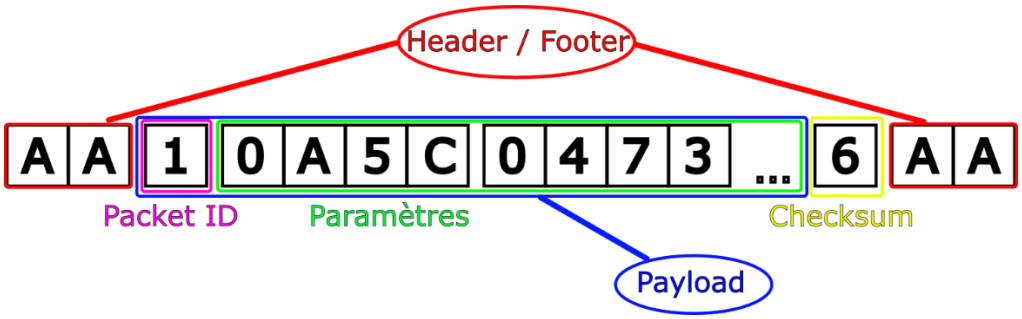


Figure 23 : Structure des paquets de données envoyés par Bluetooth

Nous aurions pu utiliser de nombreuses autres manières pour transmettre les données, et nous savons que celle-ci n'utilise pas la place de chaque caractère de manière optimale (gaspillage d'environ 4 bits de données par caractère transmis). L'objectif n'était pas d'inventer un nouveau mode de communication, mais d'en mettre un en place qui fonctionne et dont nous étions les auteurs.

Nous avons ensuite créé une fonction `BluetoothHandler::Tick()` appelée à chaque itération de la fonction `loop()`, qui se charge de lire le flux de données provenant de l'interface en série occupée par le module HC-06. Si un paquet de données est complet, alors la fonction qui doit le gérer est choisie d'une liste grâce à l'identifiant du paquet. Si cette fonction retourne une valeur `false`, un paquet de données signalant l'erreur est renvoyée à l'ordinateur. Sinon, un paquet de données signalant le succès du traitement est envoyé à sa place. La transcription de cette fonction en C++ est disponible en Annexe 5.

Avec le format actuel, nous avions à notre disposition un total de 16 types de paquets de données différents (un pour chaque caractère en hexadécimal), desquels nous en avons utilisé que sept et que nous allons aborder en détails dans la partie suivante.

E. Développement d'une solution : DobotCity

Avec tous les systèmes fonctionnels indépendamment et un système de communication permettant le partage de données entre ceux-ci, il est finalement temps d'aborder le développement de la procédure qui donnera l'image finale de notre projet.

i. Visualisation des repères de coordonnées

Même si nous commençons à voir le bout des étapes majeures de ce projet, celles restantes sont cruciales et la première que nous allons voir maintenant est la notion de repère. Au total, cinq référentiels peuvent être considérés pour notre projet :

- Le référentiel de la feuille de stockage : $\boldsymbol{R_1}$
- Le référentiel de la ville : $\boldsymbol{R_{1'}}$
- Le référentiel de la Caméra : $\boldsymbol{R_2}$
- Le référentiel du Dobot 1 : $\boldsymbol{R_3}$
- Le référentiel du Dobot 2 : $\boldsymbol{R_{3'}}$

L'objectif est d'être capable de déplacer toutes les coordonnées depuis $\boldsymbol{R_1}$, $\boldsymbol{R_{1'}}$ et $\boldsymbol{R_2}$ vers $\boldsymbol{R_3}$ ou $\boldsymbol{R_{3'}}$ si nous souhaitons que nos robots parviennent à aller chercher des blocs de manière précise. Pour mieux visualiser

chacun de ces repères, nous avons laissé en Annexe 6 une illustration réalisée sur InkScape, le logiciel de dessin vectoriel que nous avons utilisé pour créer le plan de notre ville.

Le passage de repère $\boldsymbol{R_2}$ à $\boldsymbol{R_1}$ est assez facile à réaliser. En effet, nous pouvons mesurer les distances de l'origine du repère $\boldsymbol{R_2}$, qui n'est autre que le centre du marqueur Aruco inférieur gauche délimitant la zone de stockage. En se référant à l'Annexe 6, on voit que l'axe $\boldsymbol{y_2}$ correspond à l'axe $\boldsymbol{x_1}$ dans $\boldsymbol{R_1}$, et que l'axe $\boldsymbol{x_2}$ est orienté dans le sens inverse à $\boldsymbol{y_1}$. Aussi, si $\boldsymbol{O_2}$ est l'origine de $\boldsymbol{R_2}$, un point $(\boldsymbol{M_x}, \boldsymbol{M_y})$ dans ce dernier aura pour coordonnées $(\boldsymbol{x_{O_2}} + \boldsymbol{M_y},\ \boldsymbol{y_{0_2}} - \boldsymbol{M_x})$ dans le repère $\boldsymbol{R_1}$

De cette manière, nous pouvons transmettre en Bluetooth les coordonnées des blocs directement dans le repère R_1 . Le dernier changement de repère pour passer de R_1 à R_3 sera réalisé sur la carte Arduino, grâce à la méthode détaillée dans le paragraphe suivant.

La transition vers le repère R_3 n'est pas aussi simple. En effet, si nous considérons que les robots seront toujours au même endroit, avec la même orientation, alors nous pouvons utiliser la même méthode que celle du paragraphe précédent. Cependant, il est impossible d'avoir constamment un positionnement parfait des robots, et la moindre erreur peut causer des imprécisions à l'échelle entière de la procédure de construction. C'est pour cette raison que nous avons introduit ce que l'on appelle un point de calibration, représenté par un point rouge sur chacune des feuilles.
En plaçant la tête du robot sur ce point de référence aux coordonnées connues dans le repère $\boldsymbol{R_1}$ (ou $\boldsymbol{R_{1'}}$), nous pouvons récupérer les coordonnées de ce point dans le repère $\boldsymbol{R_3}$ (ou $\boldsymbol{R_{3'}}$) associé au Dobot grâce au paquet "GetPose", qui retourne les coordonnées de la tête du Dobot dans son repère. En créant effectivement un vecteur de transition entre le repère $\boldsymbol{R_1}$ et $\boldsymbol{R_3}$ (ou $\boldsymbol{R_{1'}}$ et $\boldsymbol{R_{3'}}$), il est par la suite possible d'établir les coordonnées relatives de n'importe quel point dans ces deux repères. Cette opération nous permet notamment de corriger les erreurs de placement de la base des Dobots.
Des exemples d'angles pour deux placements différents sont donnés dans la figure ci-dessous. Ces différences d'angles se répercuteront sur le vecteur de déplacement du repère $\boldsymbol{R_1}$ vers $\boldsymbol{R_3}$.

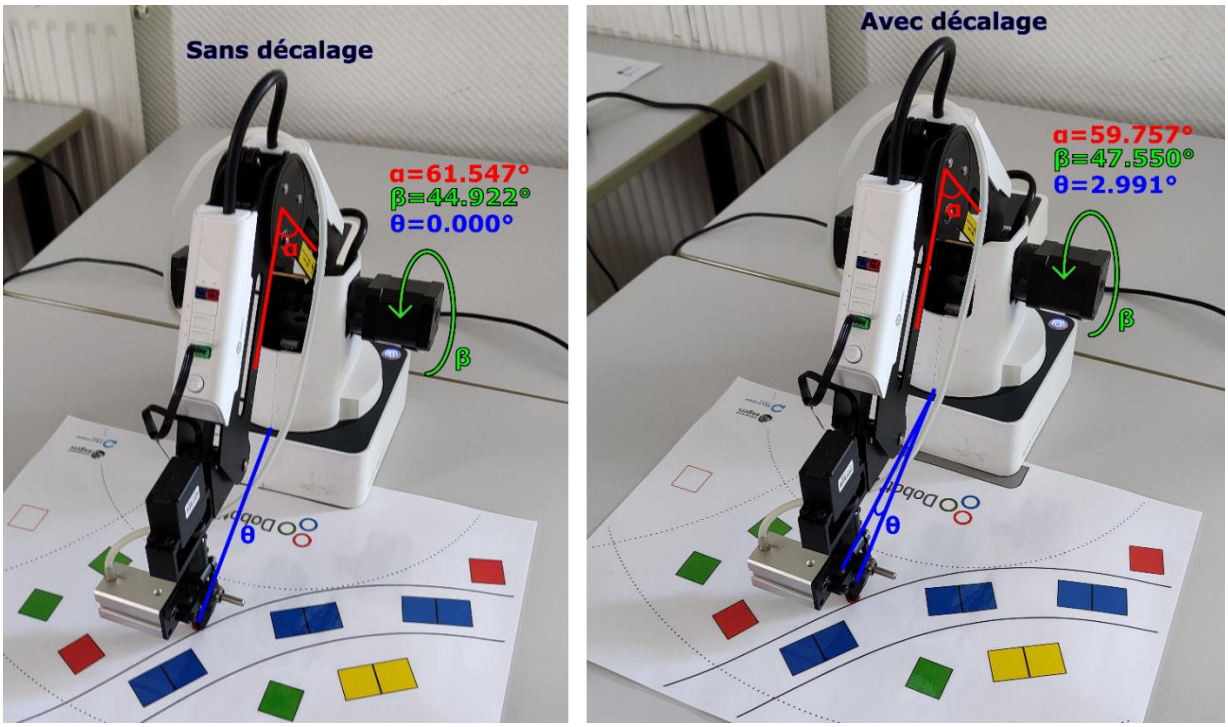


Figure 24 : Mise en évidence de la correction apportée après la calibration des Dobots

Une erreur de notre part sur cette méthode est que nous avons placé un point de calibration pour chaque Dobot. Cela permet de très bien corriger les imprécisions de placement du Dobot dans le repère R_1 ou $R_1{}'$. En revanche, nous ne pouvons en aucun cas utiliser ces informations pour déduire une position relative entre les deux robots. Aussi, nous aurions dû placer un unique point de calibration dans la zone commune aux deux Dobots, de laquelle il aurait été possible de corriger à la fois les imprécisions de placement des Dobots et celles liées à la position relative des repères R_1 et R_1' (Positionnement des feuilles).

ii. Arrangement et positionnement des composants de la ville

Comme nous l'avons abordé précédemment, nous avons utilisé l'application de dessin vectoriel InkScape pour créer le plan de notre ville. Certaines structures comme les maisons (rouge) et les arbres (vert) ont une taille de 1 bloc de largeur pour deux blocs de hauteur, tandis que d'autres comme les immeubles (jaune) et les voitures (bleu) ont une largeur de 2 blocs. Ainsi, il nous a fallu faire quelques calculs pour déterminer dans le repère R1' les coordonnées X et Y de chaque composant pour chaque structure, en ne connaissant que leur angle de rotation et les coordonnées dans ce même repère du coin supérieur gauche.

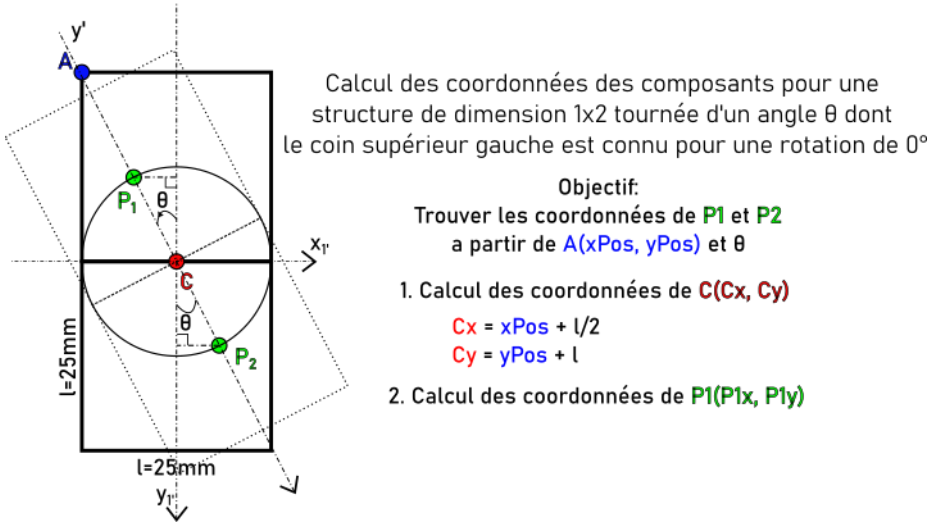


Figure 25 : Schématisation d'une structure de dimensions 1x2 avec une rotation de θ

Du schéma ci-dessus, l'on peut déduire les coordonnées du vecteur $\overrightarrow{CP_1}$, et suivre un raisonnement similaire pour $\overrightarrow{CP_2}$:

$$\cos\theta = -\frac{\overrightarrow{CP1}_y}{\frac{l}{2}} \Rightarrow \overrightarrow{CP1}_y = \frac{-l * \cos\theta}{2}$$

$$\sin\theta = -\frac{\overrightarrow{CP1}_x}{\frac{l}{2}} \Rightarrow \overrightarrow{CP1}_x = \frac{-l * \sin\theta}{2}$$

Au niveau du code, nous pouvons garder en mémoire le nombre de blocs placés pour chaque structure grâce à un entier progress. Voici ce que les formules précédentes deviennent lorsqu'elles sont traduites en C++ :

```
void GetRealCurrentPosition(float* oX, float* oY, float* oZ)
{
   // Coordonnées du centre
   float center[2] = {this->xPos + (float) UNIT_X_SIZE/2, this->yPos +
(float) UNIT_Y_SIZE};
   // Bloc gauche ou droit
   int coeff = this->progress % 2 == 1 ? -1 : 1;
   // Rotation : Degrés en radians
   float rotation = this->rot * 71.0f / 4068.0f;

   *oX = center[0];
   *oY = center[1];
   if(this->dimY > 1)
   {
      // Si la structure a une largeur de 2 blocs, alterner entre le bloc
gauche et le bloc droit
      *oX += coeff * UNIT_X_SIZE * sin(rotation) / 2.f;
      *oY += coeff * UNIT_Y_SIZE * cos(rotation) / 2.f;
   }
   // Élévation d'un bloc à chaque fois qu'un étage est fini de construire
   *oZ = floor(this->progress / (this->dimX * this->dimY)) * UNIT_Z_SIZE;
}
```

Maintenant que nous disposons de toutes les coordonnées dont nous avons besoin, c'est-à-dire de celles de chacun des composants des bâtiments de la ville, ainsi que celles des blocs présents dans la zone de stockage, nous pouvons créer une fonction compute_dobot_coordinates. Cette fonction nous permet de passer du repère $\boldsymbol{R_1}$ ou $\boldsymbol{R_{1'}}$ au repère $\boldsymbol{R_3}$ (ou $\boldsymbol{R_{3'}}$) grâce aux formules suivantes :

$$x_{R3} = x_{R1} - x_{calib} + x_{dobot\ calib}$$
$$y_{R3} = -y_{R1} + y_{calib} + y_{dobot\ calib}$$

Où (x_{R1}, y_{R1}) est le point destination dans le repère R_1, (x_{calib}, y_{calib}) les coordonnées du point de calibration dans R_1 et $(x_{dobot\ calib}, y_{dobot\ calib})$ celles du point de calibration dans le repère R_3 du Dobot.

D'un point de vue du code, ces formules peuvent être traduites de la manière suivante :

```
void compute_dobot_coordinates(int dobotId, float *x, float *y, float *z)
{
   *x = *x - calibrationPoint[dobotId][0] + dobotCalib[dobotId][0];
   *y = -*y + calibrationPoint[dobotId][1] + dobotCalib[dobotId][1];
   *z = *z + UNIT_Z_SIZE - 5;
}
```

iii. Création de la procédure des Dobots

Maintenant que tous les calculs sont faits, nous pouvons écrire la procédure pour chaque robot. Premièrement, nous devons tester que toutes les conditions sont valides pour qu'un robot puisse bouger. Les tests sont les suivants :

1. Les deux Dobots ont-ils été calibrés ?
2. Le mode Construire a-t-il été enclenché ?
3. Est-ce qu'un Dobot est déjà en train de réaliser sa procédure ?
4. La construction de la ville est-elle déjà terminée ?
5. Avons-nous reçu des données récentes de la part de l'ordinateur ?
6. Enfin, il faut vérifier le contenu de la zone de transition. Si elle est vide, on demande au Dobot 1 de faire sa procédure. Sinon, c'est au tour du Dobot 2.

Toutes ces conditions, *hormis la troisième*, sont faciles à vérifier, puisqu'il ne s'agit que de simples valeurs booléennes qui sont mises à jour en fonction des données reçues de l'utilisateur et du statut d'exécution du programme sur la carte Arduino. Ce qui aurait pu nous poser un problème est donc de vérifier si un autre Dobot est en train de réaliser sa procédure. La manière la plus simple que nous avons trouvée pour répondre à ce besoin est de rajouter, en plus des connexions **RX** et **TX**, un troisième fil entre la sortie 5V **EIO18** des Dobots et un pin de l'Arduino. Ainsi, nous pouvons ajouter à la procédure de chaque Dobot l'activation de son pin **EIO18** au début de sa procédure, et sa désactivation à la fin. Pour vérifier si un Dobot est déjà en train de travailler, nous n'avons alors qu'à regarder l'état de son pin **EIO18**, et si l'un d'entre eux est actif, alors nous savons qu'il faut attendre.

Voici donc la procédure pour chacun des Dobots :

- Dobot 1 : Approvisionnement
    - Entre tous les blocs envoyés par l'ordinateur en Bluetooth, choisir le premier du même type que la structure actuellement en construction. S'il n'y en a pas de disponible, s'arrêter là et attendre
    - Effectuer le changement de repère des coordonnées du bloc choisi (du repère **R1** au repère **R3)**
    - Activer la sortie EIO18
    - Ouvrir la pince
    - Aller aux coordonnées calculées, avec le mode de mouvement JUMP (pour arriver verticalement par rapport au bloc) et la rotation transmise par Bluetooth
    - Fermer la pince
    - Attendre 500 ms pour s'assurer que la pince a eu le temps de se fermer
    - Emmener le bloc dans la zone de transition avec une rotation de 0°, en utilisant le mode JUMP pour ne pas déranger les autres blocs dans la zone de stockage
    - Ouvrir la pince
    - Retourner à droite de la zone de stockage pour libérer l'espace pour le Dobot 2
    - Désactiver la sortie EIO18

- Dobot 2 : Construction de la ville
    - Récupérer les informations de la structure actuellement en construction
    - Ouvrir la pince
    - Déplacer les coordonnées de la destination du repère **R1'** au repère **R3'**
    - Activer la sortie EIO18
    - Se déplacer vers la zone de transition avec une rotation de 0°, en utilisant le mode JUMP pour arriver verticalement par rapport au bloc
    - Fermer la pince
    - Attendre 500 ms pour s'assurer que la pince a eu le temps de se fermer
    - Se déplacer aux coordonnées de la destination dans le repère **R3'**, avec la même rotation que celle de la structure. Utiliser le mode JUMP pour passer au-dessus des structures déjà construites.

- Ouvrir la pince
- Revenir à la position initiale
- Désactiver la sortie EIO18

Une transcription de ces deux procédures en C++ avec la bibliothèque DobotNet est disponible en Annexe 7. Après réflexion, nous avons remarqué plusieurs éléments qu'il serait possible de changer pour une procédure plus fluide et rapide. En effet, le mode de déplacement JUMP cause souvent des déplacements inutiles. Même s'il faut s'assurer de ne pas rentrer en collision avec des structures déjà construites, il est possible à plusieurs reprises d'effectuer deux mouvements linéaires au lieu des trois imposés par la méthode JUMP.
De plus, l'attente qu'un Dobot soit revenu à sa position initiale pour que l'autre puisse débuter sa procédure permet d'assurer qu'ils n'entreront jamais en collision. En revanche, il serait possible de décomposer le retour à la position initiale en deux mouvements. Nous pourrions ainsi signaler à la fin du premier mouvement que l'autre Dobot peut commencer sa procédure et terminer le retour en parallèle.

Pour informer l'utilisateur de l'état de la procédure du côté de l'Arduino, nous avons également pensé qu'il serait intéressant d'intégrer un écran LCD 4x20 à notre projet (4 lignes de 20 caractères). Grâce à un microcontrôleur PCF8574 et un potentiomètre pour le contraste intégré à l'écran, les connexions de celui-ci à la carte Arduino sont très simples, avec l'alimentation grâce aux pins GND et VCC, et l'interface I2C avec les pins SCL et SDA. Contrairement à l'interface UART où les communications sont régulées par un baud rate (nombre de bits par seconde) et une vérification de l'intégrité des données, une interface I2C est régulée par un signal d'horloge, ou chaque changement d'état signifie qu'un nouveau bit de donnée peut être lu. Après la modification du code pour afficher du texte à l'écran, voici deux exemples des informations que nous pouvons afficher à l'utilisateur :

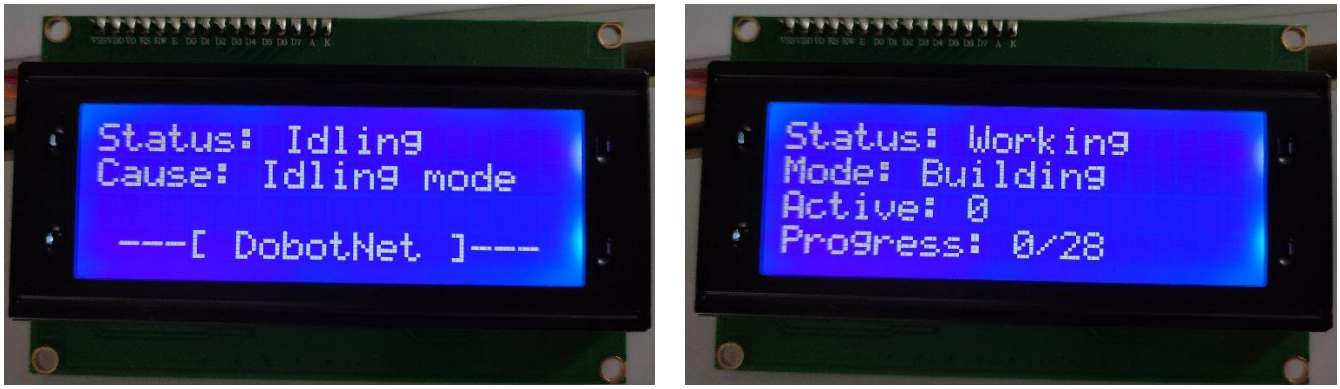


Figure 26 : Exemples de messages affichés par l'écran LCD 4x20

Il est possible de trouver l'écran dans d'autres états, notamment pour indiquer qu'un des Dobots n'est pas calibré, ou encore si le module Bluetooth n'a pas reçu de paquet de présence de la part de l'ordinateur en 5 secondes d'intervalle.

iv. Implémentation d'une interface utilisateur (GUI)

Dans une idée d'amélioration de l'expérience utilisateur, nous avons adapté notre code python pour l'intégrer à une belle interface de contrôle. Elle permet notamment de visualiser en temps réel le réglage des paramètres de la fonction `cv2.Canny()` à l'aide de deux curseurs :

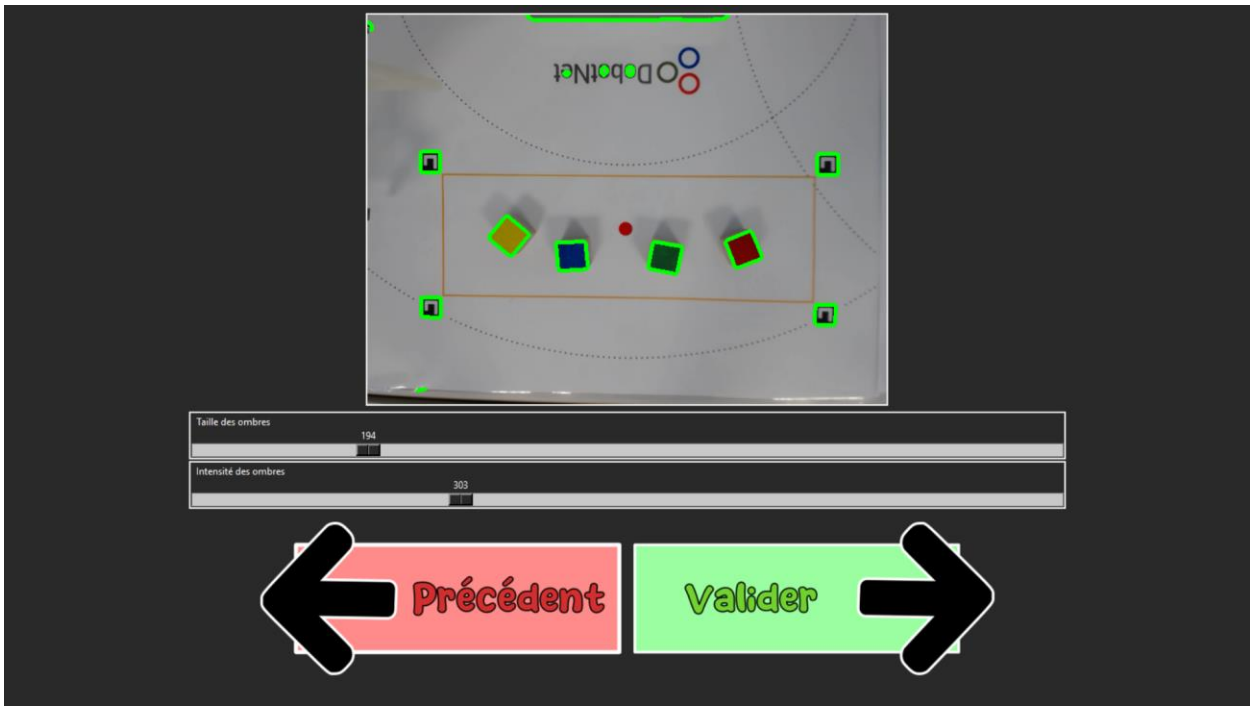


Figure 27 : Interface de calibration des paramètres pour l'algorithme de Canny

Nous y avons également ajouté des boutons pour simplifier les interactions avec le module Bluetooth, ainsi qu'un affichage du flux de la caméra une fois le traitement d'image effectué, de sorte à mieux visualiser les résultats de la détection des blocs :

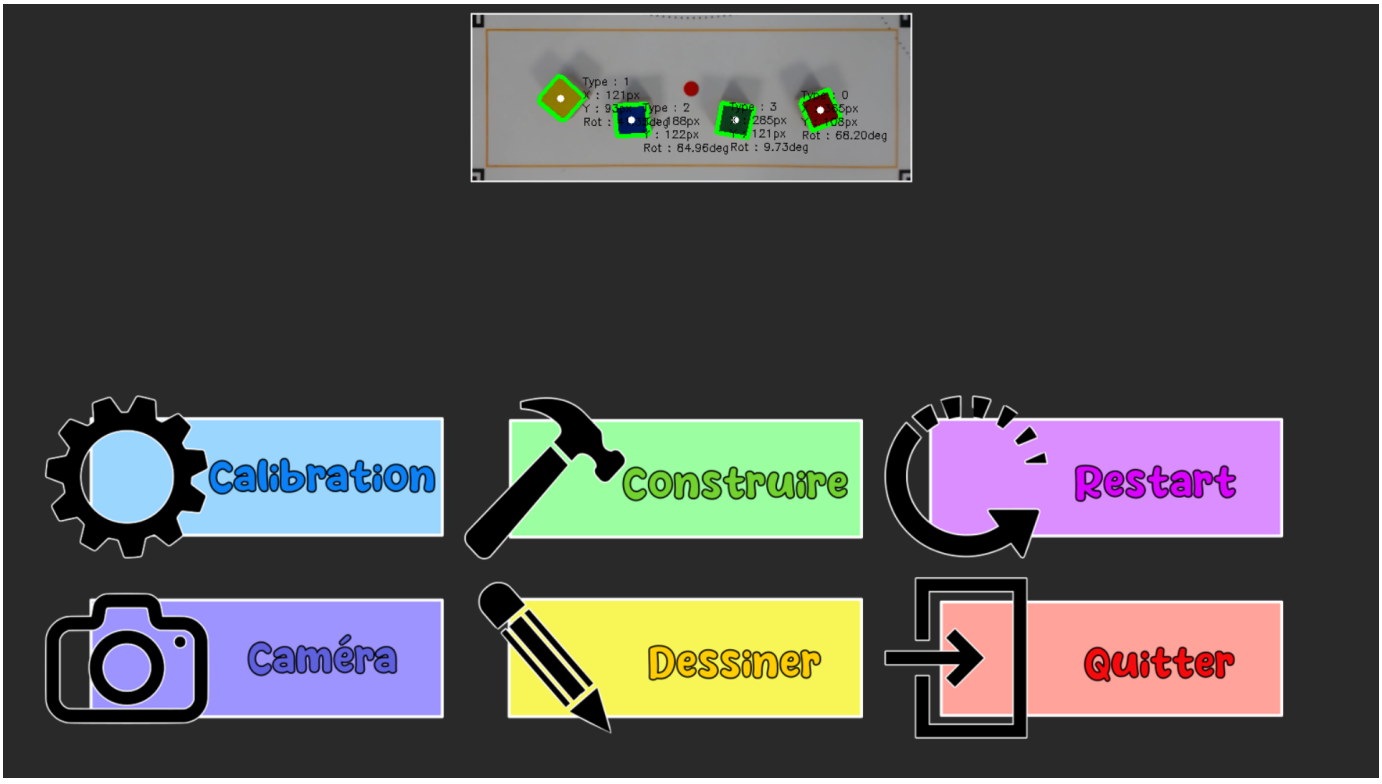


Figure 28 : Interface de contrôle pour communiquer facilement avec la carte Arduino

Nous avons ensuite inséré dans le code Arduino des fonctions pour gérer les paquets de données envoyés par chacun des boutons ci-dessus. À titre d'exemple, le bouton **Construire** envoie les données : "`AA212AA`", que l'Arduino interprète comme un changement de mode vers celui de "Construction". Un autre exemple est le bouton **Reset** qui envoie le paquet : "`AAE3AA`", demandant à l'Arduino d'effacer sa mémoire concernant l'état de construction de la ville, et de se mettre en mode pause.

Le bouton **Dessiner** avait pour but de laisser l'utilisateur créer sa propre ville grâce à une interface sur l'ordinateur. Tandis que l'ajout de cette fonctionnalité du côté du code Arduino est très simple (tous les systèmes de communication et de gestion de la ville le permettent déjà), nous n'avons pas eu le temps de créer une interface de dessin avec Tkinter.

F. Conception d'une carte électronique

La partie logicielle qui représente une grosse portion de notre projet étant terminée, il nous restait alors à peaufiner les derniers détails pour finaliser la première version. Avec les ajouts cités précédemment pour donner un aspect plus agréable visuellement de notre projet (*interface de contrôle*, *écran LCD*, …), quelque chose restait clairement à améliorer : la connectique entre les différents composants. En effet, actuellement, tous les composants du projet (*module Bluetooth*, *écran LCD*, *Dobots*, *résistances*…) sont connectés à l'aide de fils et d'un breadboard.

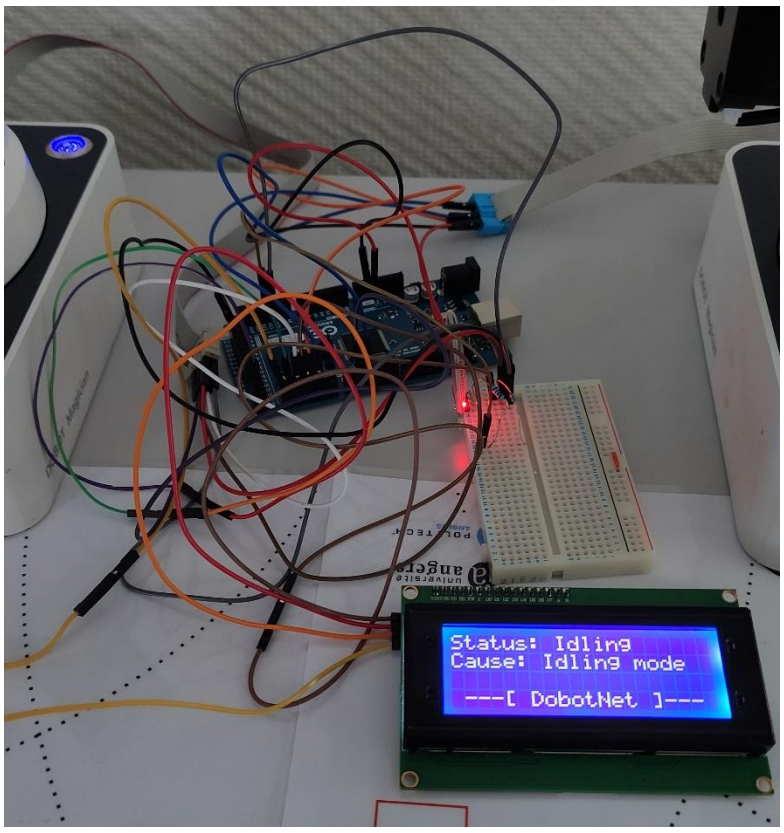


Figure 29 : Apparence actuelle des connexions entre les éléments de notre projet

Ce genre de câblage est bien au début pour faire des tests, mais nous avons découvert en parlant avec les membres d'autres projets qu'il est possible, maintenant que notre projet est finalisé, de se débarrasser de la majorité de ces fils grâce à une carte électronique. C'est alors que nous nous sommes renseignés pour créer notre propre shield (extension) pour Arduino Mega. Ce ne fut cependant pas une mince à faire, car avant de

pouvoir le fabriquer, il était nécessaire de se familiariser avec un logiciel qui nous était jusqu'alors complètement inconnu : EAGLE par AutoDesk. Néanmoins, nous avons regardé plusieurs vidéos de démonstration ainsi que suivi les conseils de plusieurs de nos camarades et après quelques essais (notamment pour retirer les collisions entre les fils), nous sommes parvenus à en créer une première version dont vous pouvez voir le schéma électrique ainsi que l'arrangement des connexions ci-dessous :

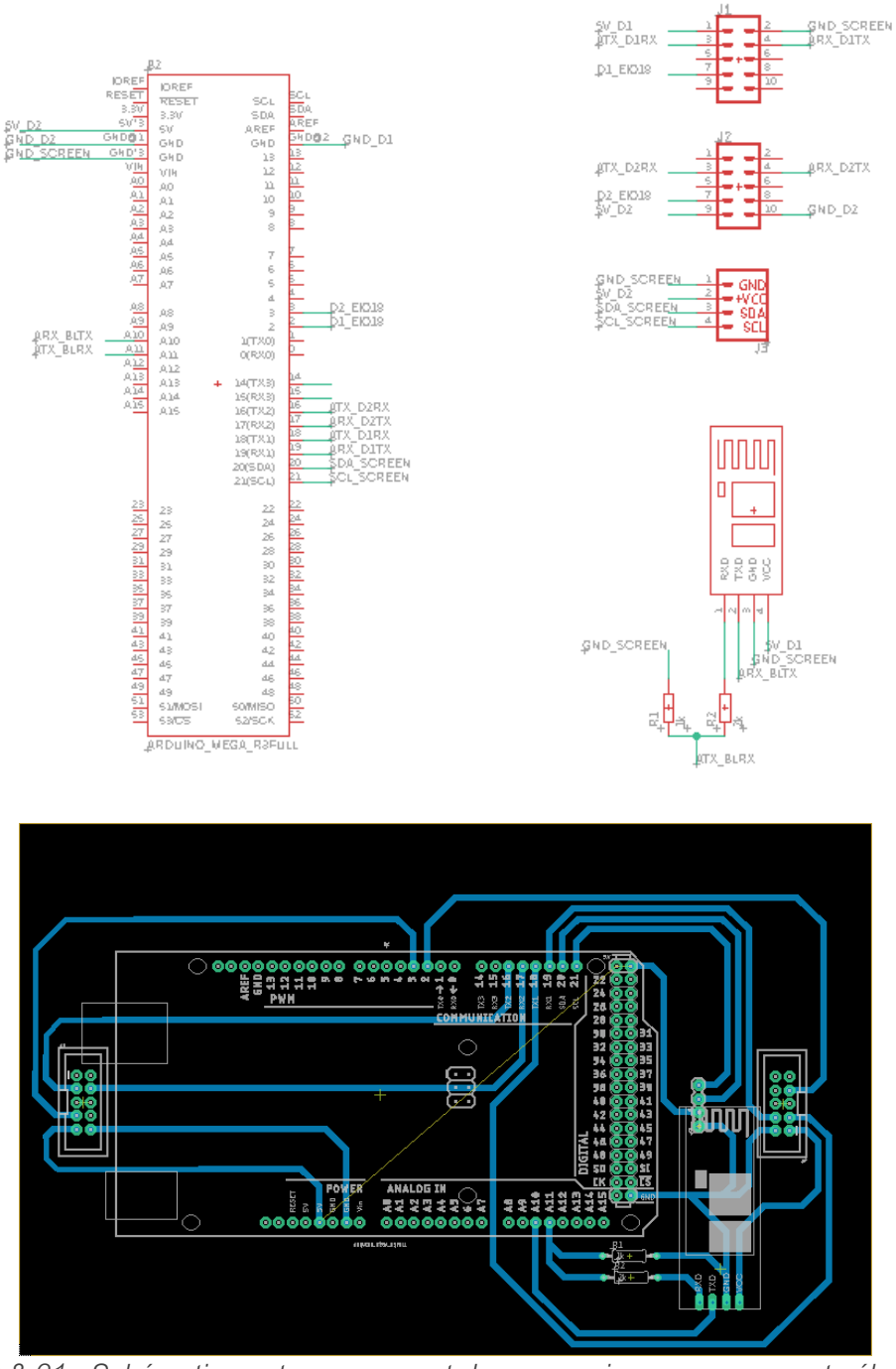


Figure 30 & 31 : Schématique et arrangement des connexions pour une carte électronique

Chaque connecteur 10 pins (1 et 2) sur les schémas ci-dessus représente un Dobot connecté à la carte Arduino, de la manière vue dans la partie V. B) ii. On retrouve également des emplacements pour les connexions de l'écran (3) et du module HC-06 (4).

Nous n'aurons malheureusement pas le temps d'imprimer cette carte avant la fin des projets, mais nous laissons les fichiers nécessaires au cas où de futurs étudiants reprendraient là où nous nous sommes arrêtés pour apporter leurs améliorations.

G. Modélisation d'un boîtier sur SolidWorks

Un shield (extension) pour notre Arduino Mega aurait, certes, amélioré l'apparence visuelle de l'ensemble, mais nous avons aussi pensé qu'il serait encore mieux de créer une boîte pour contenir les éléments de notre projet. Elle nous permettrait alors de mettre en évidence l'écran d'information, qui a pour but d'aider l'utilisateur, en cachant tous les éléments qui fonctionnent en arrière-plan tels que le module Bluetooth et la carte Arduino. Après des mesures méticuleuses pour s'assurer que l'espace à l'intérieur est suffisant pour accueillir l'ensemble des composants, nous avons créé le prototype présenté en Annexe 8 en le dessinant sur SolidWorks.

En revanche, tant que nous n'avons pas imprimé la carte électronique, nous ne pouvons pas prendre le risque d'imprimer cette boîte puisque les dimensions de cette dernière pourraient changer, menant à un gaspillage de temps et de matériaux. Comme pour les fichiers EAGLE, nous laissons donc le modèle 3D dans les fichiers de notre projet, au cas où d'autres étudiants souhaiteraient le continuer et le finaliser.

VI. REMARQUES SUR LA TECHNIQUE

A. Erreurs d'organisation

Bien que nous ayons anticipé une grande partie des problèmes potentiels qui auraient pu nous arriver en préambule de notre projet, tout au long de ce dernier, et même une fois l'avoir achevé, nous nous sommes rendu compte que nous avions omis certains d'entre eux. Il est évident que nous ne pouvons pas tous les développer dans le cadre de ce rapport, mais il nous semble important de revenir sur quatre d'entre eux particulièrement. Il ne s'agit pas toujours de problèmes techniques et d'ailleurs, le premier dont on souhaite parler est celui portant sur les erreurs d'organisation que l'on a pu commettre. Cela peut paraître étonnant avec les outils mis en place tels qu'un serveur Discord dédié au projet ou encore le tableau partagé Trello évoqué précédemment, mais il est vrai que nous avons manqué de rigueur quant à leur utilisation. Le souci n'est pas la catégorisation des tâches que nous avions faites et qui, pour le coup, était très efficace, mais il se trouve plutôt au niveau du temps accordé à quelques-unes d'elles.

En effet, il nous est parfois arrivé de nous focaliser sur un élément qui n'était pas nécessairement important à un moment donné, à défaut de se concentrer sur ce qui l'était vraiment. Pour donner un exemple, nous avons parfois marqué comme terminé plusieurs parties de notre projet que nous trouvions moins intéressantes que d'autres, sans pour autant avoir vérifié entièrement leur fonctionnement. C'était notamment le cas pour l'intégration des changements de repères dans le code où, puisque nous avions obtenu un résultat cohérent dans des conditions presque parfaites, nous en avions déduit que tout fonctionnait. En revanche, jusqu'à quelques semaines avant la fin des projets, cette erreur nous a constamment causé de grosses imprécisions au niveau du calcul des coordonnées des blocs et des bâtiments dans le repère des Dobot. Ce n'est qu'après avoir perdu plusieurs heures à travailler sans relâche sur la résolution des imprécisions que nous avons pensé à revoir ces calculs, qui étaient légèrement faux depuis plusieurs mois.

Nous avons aussi eu un problème similaire concernant la création du plan de la ville sur InkScape. Cette étape aurait dû être complétée vers le milieu du projet, et pourtant, nous avions déjà une première version seulement trois semaines après le début des projets. Elle nous paraissait plus amusante à faire que d'autres, et nous avons décidé de passer beaucoup de temps dessus dès le début, ce qui nous a mené à des pertes de temps considérables dans le futur puisque nous n'avions pas encore une bonne compréhension du champ d'accessibilité des Dobots, du temps qu'il faudrait pour construire de telles villes, etc.
Voici la comparaison entre notre première version de la carte, et celle que nous avons décidé d'adopter :

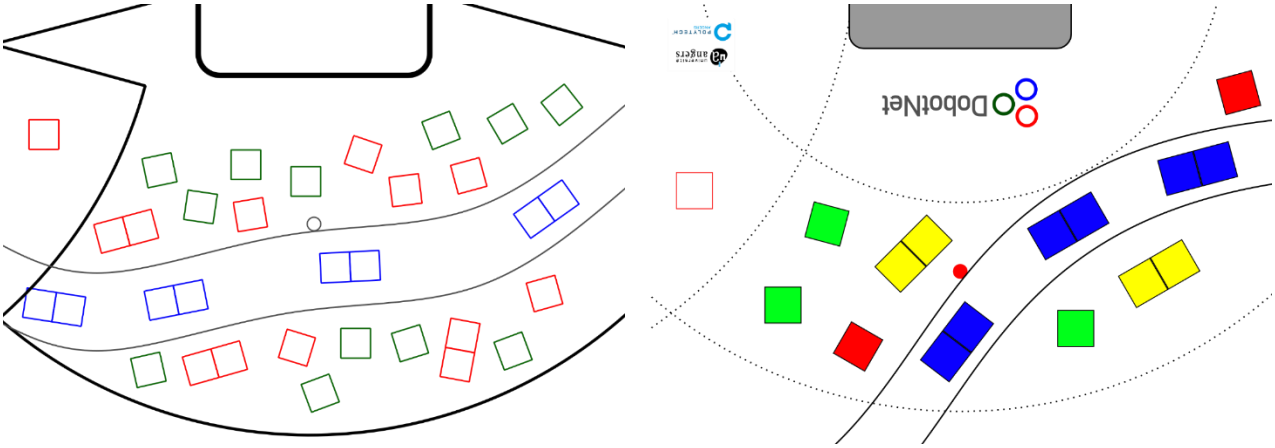


Figure 32 : Comparaison entre la première et la dernière version du plan de la ville

Un dernier point sur lequel nous aurions dû porter plus d'attention est la création de programmes séparés pour tester efficacement des parties isolées de notre projet. Nous avons perdu une quantité de temps importante sur différents aspects du projet. Que se soit lors des essais des fonctions d'OpenCV sans avoir créé au préalable une interface qui nous permettrait de voir le résultat en temps réel, ou encore lors de la conception des procédures de chaque Dobot où il fallait attendre que l'autre termine son cycle avant de voir le résultat, nous pouvons dire que nous n'avons pas été des plus efficaces. Le jour où nous avons finalement décidé de créer une interface d'essais et de visualisation, il ne nous a fallu que quelques heures pour trouver une combinaison d'algorithmes qui produisait un résultat constant et acceptable.

B. Estimation des délais

Une autre erreur que nous avons régulièrement commise tout au long de ce projet est la sous-estimation du temps nécessaire pour compléter une étape.

D'ailleurs, en mentionnant la notion de temps, c'est quelque chose qu'il a failli nous manquer vers la fin du projet et pour cause, nous avons pas mal sous-estimé le temps nécessaire à la réalisation de chaque étape. Ce n'est pas toujours évident de connaître à l'avance la durée d'une tâche portant sur des domaines auxquels nous ne nous sommes jamais confrontés tels que la robotique ou encore l'analyse et le traitement d'une image. Ce manque de connaissances nous a donc quelquefois induits en erreur en ne prenant pas systématiquement une marge de temps suffisante au cas où l'on rencontrerait des difficultés. Le problème étant que lorsque cela se produit, les heures perdues sur une tâche qui à la base nous semblait rapide se répercutent sur l'étape qui est censée suivre. Qui plus est, il est assez récurrent lorsque l'on rédige un programme dans un langage complexe et auquel nous sommes peu habitués, comme le C++, d'essayer de comprendre des exemples, d'oublier certaines notations, de chercher des solutions complexes alors qu'une autre méthode nous permettrait d'obtenir le même résultat plus rapidement. Ce sont des choses auxquelles nous aurions dû penser pour prévoir du temps supplémentaire et ne pas négliger les tests qui s'en suivent.

C. Confiance dans le matériel

La dernière erreur à laquelle nous avons fait face est enfin due à une trop grande confiance dans le matériel mis à notre disposition et dont nous n'avons pas suffisamment pris en compte les incertitudes. Que ce soit au début du projet ou à la fin, cela nous a fait défaut.

Parmi les outils dont nous disposions, on peut d'abord porter un œil sur les Dobots. Ces derniers disposent de nombreuses compétences comme nous avons pu le voir au cours de ce rapport. Néanmoins, la documentation fournie par les concepteurs des Dobots Magician était parfois incomplète ou manquait de précision. De plus, certains des protocoles intégrés à la démonstration officielle sur laquelle nous nous sommes basés au début du projet étaient faux. Il en résulte alors des difficultés pour donner des instructions aux robots si l'on ne connaît pas les paquets de données permettant de les faire se mouvoir, prendre des objets, tourner l'outil qu'ils utilisent, etc. C'est aussi une des raisons qui nous a donné l'envie de faire notre propre solution, pour corriger les erreurs et donner un accès facile au protocole de communication des robots.

Cependant, il y a une autre source d'incertitude bien plus conséquente que nous avons connue au moment de la détection des blocs : il s'agit de la caméra. Bien qu'elle soit un atout majeur du projet, elle est à l'origine d'une grande partie de l'imprécision qui en résulte. Les raisons qui mènent à cela sont multiples, mais la principale est sa position et son orientation relative à la feuille de la zone de stockage. Ces degrés de liberté rendent très complexe la conversion exacte de coordonnées en pixels d'une image 2D en coordonnées dans l'espace.

VII. BILAN

Ce projet a été une expérience extrêmement enrichissante dans son ensemble. Nous sommes ravis d'avoir été attribués à une initiative aussi complète, qui nous a permis d'explorer de nombreuses nouvelles notions passionnantes et utiles pour le futur, allant de la robotique à l'électronique en passant par la vision par ordinateur.

Lors de ce travail en binôme, nous avons réussi à mettre en place une organisation, en listant des tâches à faire et en se les attribuant pour se répartir le travail. Au long de ces six derniers mois, nous avons fait preuve d'un esprit collaboratif, en s'entraidant dans les domaines que l'on ne maîtrisait pas, et en acquérant de nouvelles connaissances dans les autres. En dépit des problèmes rencontrés, que ce soit au niveau de la documentation imprécise des robots, du domaine inconnu de la vision par ordinateur, ou de la programmation en C++, nous avons réussi à mettre en place des solutions pour aujourd'hui avoir un résultat abouti dont nous sommes fiers.

Il est évident que notre technique n'était pas parfaite, nous nous en sommes rendu compte lorsque nos capacités étaient contestées par des défis stimulants. Cependant, nous tirons une leçon de nos erreurs, et ces expériences du semestre passées nous seront forcément utiles pour le futur. Nous sommes avant tout très reconnaissants de l'occasion que fut la participation à ce projet, et nous retenons ce que nous avons appris pour les prochains défis d'ingénierie que nous rencontrerons. L'objectif était avant tout de surmonter un besoin en équipe, et aujourd'hui, au vu de ce que nous avons accompli, nous pensons avoir réussi cette tâche.

VIII. LIENS EXTERNES

[1] Dobot – Site Officiel
https://www.dobot-robots.com/products/education/magician.html

[2] Trello – Page d'accueil
https://trello.com/home

[3] Interface UART
https://www.rohde-schwarz.com/us/products/test-and-measurement/essentials-test-equipment/digital-oscilloscopes/understanding-uart_254524.html

[4] Protocole de communication d'un Dobot Magician
https://www.alcom.no/wp-content/uploads/2019/11/Dobot-Communication-Protocol-V1.1.5-1.pdf

[5] Documentation sur les buffer circulaires (Ring Buffer)
https://en.wikipedia.org/wiki/Circular_buffer

[6] Bibliothèque open-source – DobotNet pour Arduino
https://github.com/MisTurtle/DobotNet

[7] Documentation sur la vision par ordinateur (Computer Vision)
https://en.wikipedia.org/wiki/Computer_vision

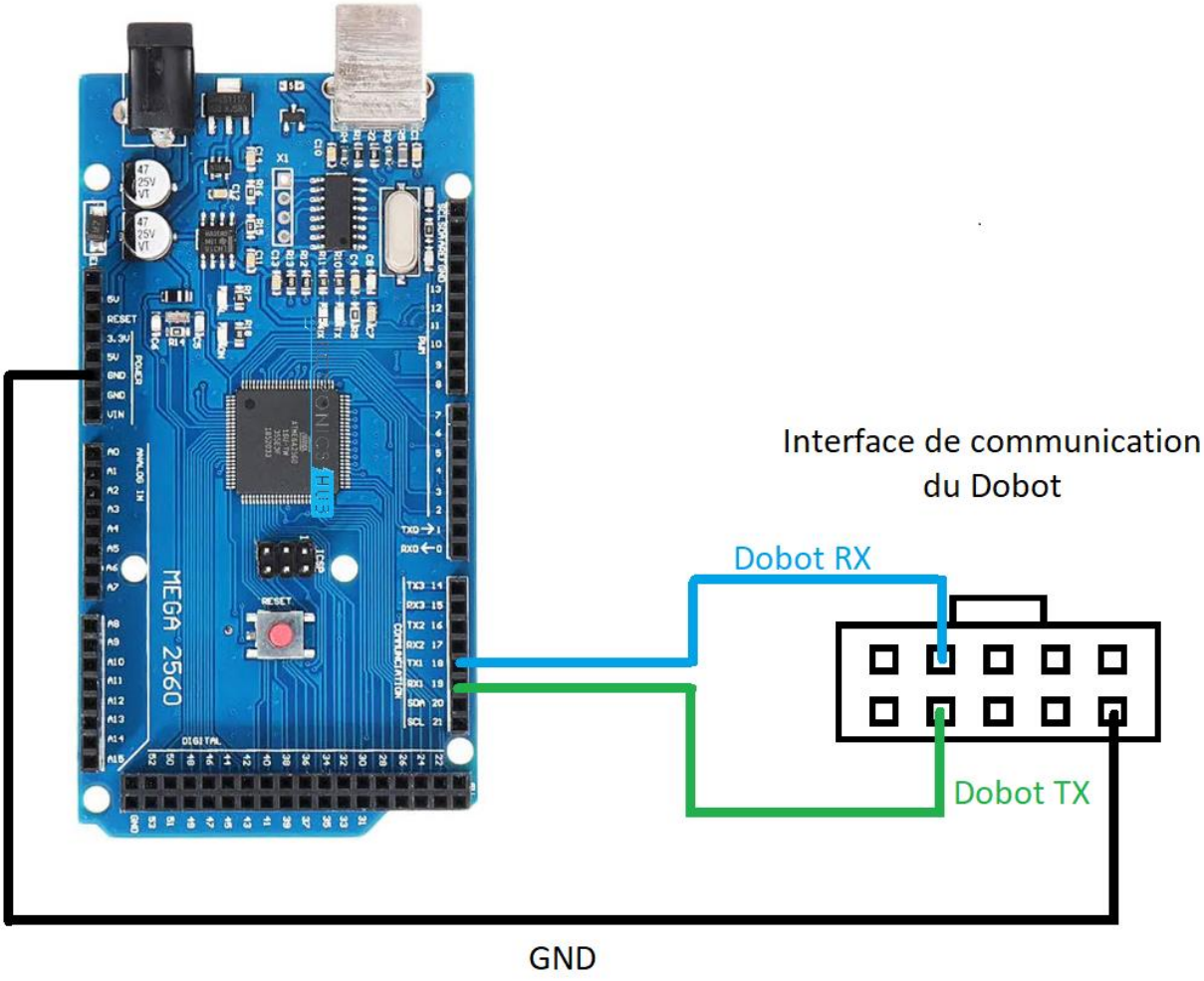
[8] Informations sur le flou gaussien (Gaussian Blur)
https://en.wikipedia.org/wiki/Gaussian_blur

[9] Article comparant deux algorithmes de détection de contour (Canny contre Sobel)
https://medium.com/srm-mic/finding-the-edge-canny-and-sobel-detectors-part-1-65a59b7ef62a

[10] Informations supplémentaires concernant le gradient d'une image
https://pyimagesearch.com/2021/05/12/image-gradients-with-opencv-sobel-and-scharr/

ANNEXE 1 : CONNEXION D'UN DOBOT A UNE ARDUINO MEGA

Le schéma ci-dessous illustre la connexion minimale pour contrôler un robot Dobot Magician depuis une carte Arduino Mega via son interface UART.



ANNEXE 2 : EXEMPLE D'UTILISATION DE DOBOTNET – 1 DOBOT

Le programme suivant permet de faire bouger linéairement un Dobot connecté aux pins RX1 et TX1 d'une carte Arduino Mega, en alternant entre les positions {x=200, y=0, z=0} et {x=300, y=0, z=0}

```
#include "DobotNet.h"

// Serial port on which the Dobot will be connected
HardwareSerialWrapper dobotSerial{&Serial1};
// Instantiate a Dobot instance of ID 0 that uses Serial 1
DobotInstance dobot{&dobotSerial, 0};
// Loop count
int count = 0;

void setup()
{
   Serial.begin(9600);
   // Initialize the dobot network with 1 dobot
   DobotNet::Init(&dobot, 1);
   // Send movement speed and acceleration parameters to the dobot
   send_movement_parameters();
}

void send_movement_parameters()
{
   dobot.SetPTPCoordinateParams(100, 100, 80, 80, true); // XYZ moving speed
   dobot.SetPTPCommonParams(50, 50, true); // Ratios

   DobotNet::Tick(nullptr);
}

void loop()
{
   float x;

   if(count++ % 2) x = 200;
   else x = 300;

   // Move to a given point using the linear moving mode
   dobot.MoveTo(MOVL_XYZ, x, 0, 50, 0);

   // Send all previously requested packets
   DobotNet::Tick(nullptr);

   // Wait 3 seconds
   delay(3000);
}
```

ANNEXE 3 : EXEMPLE D'UTILISATION DE DOBOTNET – 2 DOBOTS

Le programme suivant fait bouger linéairement deux Dobots connectés à une carte Arduino Mega. Un des Dobots doit être connecté aux pins RX1 et TX1, et l'autre aux pins 12 et 13.

```
#include "DobotNet.h"
#include <SoftwareSerial.h>

// Emulate Serial port on pins RX: 12 and TX: 13
SoftwareSerial Serial5{12, 13};
// Serial ports on which the Dobots will be connected
HardwareSerialWrapper dobotS1{&Serial1};
SoftwareSerialWrapper dobotS2{&Serial5};

// Instantiate two Dobot instances
DobotInstance dobots[2] = {{&dobotS1, 0}, {&dobotS2, 1}};

// Loop count
int count = 0;

void setup()
{
   Serial.begin(9600);
   // Initialize the dobot network with 1 dobot
   DobotNet::Init(dobots, 2);
   // Send movement speed and acceleration parameters to the dobot
   send_movement_parameters();
}

void send_movement_parameters()
{
   for(auto & dobot : dobots)
   {
      dobot.SetPTPCoordinateParams(100, 100, 80, 80, true); // XYZ moving
speed
      dobot.SetPTPCommonParams(50, 50, true); // Ratios
   }
   // Submit movement packets, without callback to be performed
   DobotNet::Tick(nullptr);
}

void loop()
{
   float x;
   if(count++ % 2) x = 200;
   else x = 300;

   // Move to a given point using the linear moving mode
   for(auto & dobot: dobots) dobot.MoveTo(MOVL_XYZ, x, 0, 50, 0);

   // Send all previously requested packets
   DobotNet::Tick(nullptr);

   // Wait 3 seconds
   delay(3000);
}
```

ANNEXE 4 : ALGORITHME DE RANGEMENT DANS LE SENS HORAIRE

Le fonction suivante, programmée en Python, permet d'ordonner une liste de quatre points dans un sens horaire.

Source : https://pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/

```
def order_points(pts):
    # Trier les points selon leur coordonée en X
    xSorted = pts[np.argsort(pts[:, 0]), :]
    # Récupérer le point le plus à gauche et celui le plus à droite
    leftMost = xSorted[:2, :]
    rightMost = xSorted[2:, :]
    # Tri des coordonnées les plus à gauche selon leur valeur en Y
    # pour determiner celui en haut à gauche et en bas à gauche
    leftMost = leftMost[np.argsort(leftMost[:, 1]), :]
    (tl, bl) = leftMost
    # Utilisation du point supérieur gauche comme reference pour
    # calculer les distances euclidiennes avec les points à droite
    # D'après le théorème de Pythagore, celui le plus éloigné sera le point
    # inférieur droit
    D = dist.cdist(tl[np.newaxis], rightMost, "euclidean")[0]
    (br, tr) = rightMost[np.argsort(D)[::-1], :]
    # Retourner les points ordonnés dans le sens horaire
    return np.array([tl, tr, br, bl], dtype="float32")
```

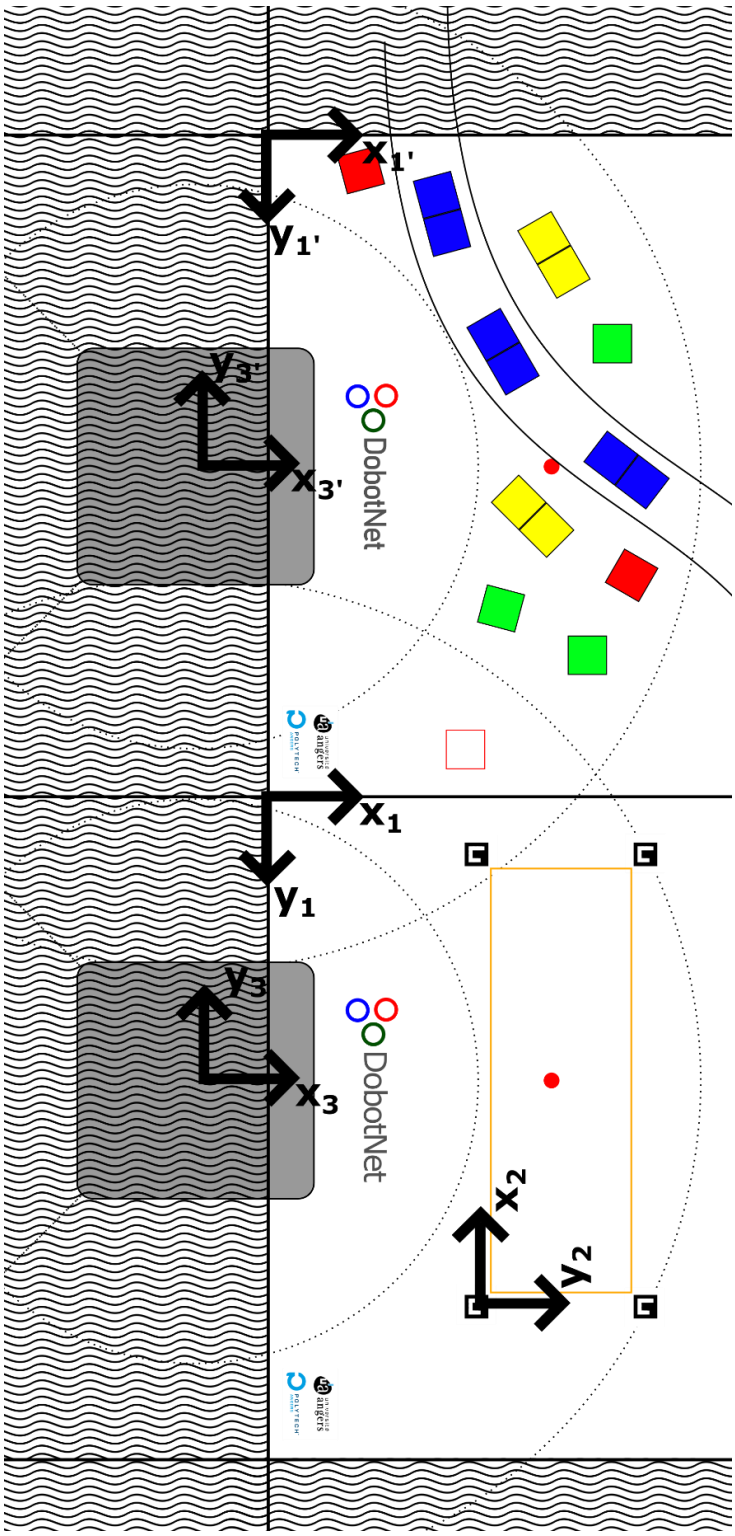
ANNEXE 5 : TRANSCRIPTION DE BLUETOOTHHANDLER ::TICK()

La fonction suivante permet de lire et de gérer les paquets de données reçus par Bluetooth du côté de la carte Arduino.

```
void BluetoothHandler::Tick()
{
   // Récupérer les données reçues par Bluetooth
   while(this->serialPort->available())
   {
   // Lire tant que des données sont disponibles et qu'un paquet n'est pas
complet
      while(this->serialPort->available() > 0 && this->payload_length <
MAX_PACKET_LENGTH && !this->IsPayloadComplete())
         this->payload[(this->payload_length)++] = (char) this->serialPort-
>read();

      // Le paquet est trop grand pour être valide
      if(this->payload_length >= MAX_PACKET_LENGTH) return this-
>ThrowPayloadError("Invalid Payload");

      // Le paquet n'est pas encore reçu entièrement
      if(!this->IsPayloadComplete()) return;

      // La checksum n'est pas bonne
      if(!this->ValidateChecksum()) return this->ThrowPayloadError("Invalid
Checksum");

      // Lecture de l'identifiant du paquet
      int pid = Hex2Int(this->payload[2]);

      // Le paquet n'est pas défini
      if(pid < 0 || pid >= MAX_PACKET_UNIQUE_IDS) return this-
>ThrowPayloadError("Invalid Packet Id");

      if(this->handlers[pid] == nullptr) return this-
>ThrowPayloadError("Unhandled Packet Id");

      // Le paquet n'a pas été traité correctement
      if(!this->handlers[pid](this->payload, this->payload_length)) return
this->ThrowPayloadError("Failed to process packet");

      // Succès, envoie de la réponse positive
      this->last_rcv_ts = millis();
      this->Acknowledge(true);
      this->ClearPayload();
   }
}
```

ANNEXE 6 : PLAN DE LA VILLE ET VISUALISATION DES REPERES

L'image ci-dessous représente le plan de la ville que nous essayons de construire, avec les différents repères importants marqués en noir.



ANNEXE 7 : TRANSCRIPTION DES PROCEDURES DES DOBOTS

Procédure du Dobot gérant la zone de stockage

```
bool PerformStorageStep(DobotBuilderUnit* bot)
{
   // Récupérer un bloc du type requis
   auto target = mapHandler.GetTargetCompound();
   if(target == nullptr)
   { // Aucun bloc trouvé
      perform_delay(WAIT_CAUSE_NO_BLOCK, mapHandler.GetTargetSlot()->type);
      return false;
   }

   // Calcul des coordonnées dans le repère du Dobot
   float x, y, z, rot;
   target->GetRealPosition(&x, &y, &rot);
   z = offset[0][2];
   compute_dobot_coordinates(0, &x, &y, &z);

   // Aller chercher le bloc
   set_dobot_working(0, true, false); // Signaler le début de la procédure
   bot->SetGripping(false, true); // Ouvrir la pince
   bot->MoveTo(JUMP_XYZ, x, y, z, -rot); // Aller à la position calculée
   bot->SetGripping(true, true); // Fermer la pince
   bot->SerialProcess(nullptr); // Envoyer les paquets

   // Aller le placer
   bot->Delay(500, true); // Attendre 500ms
   bot->MoveTo(JUMP_XYZ, computedTransitionSlot[0][0],
computedTransitionSlot[0][1], z, 0); // Aller sur la zone de transition
   bot->SetGripping(false, true); // Ouvrir la pince
   bot->Delay(500, true); // Attendre 500 ms
   bot->SerialProcess(nullptr); // Envoie des paquets

   // Retourner en position initiale
   bot->DisableGripper(true); // Désactiver la pompe
   bot->MoveTo(JUMP_XYZ, 225, -180, offset[0][2] + 5, 0);
   set_dobot_working(0, false, true); // Retour à la position d'origine
   bot->SerialProcess(nullptr); // Envoie des paquets

   // Attendre que la première instruction soit traitée par le Dobot
   wait_until_working(0, true);

   // Définir le contenu de la zone de transition
   mapHandler.SetTransitionSlotContent(target->type);
}
```

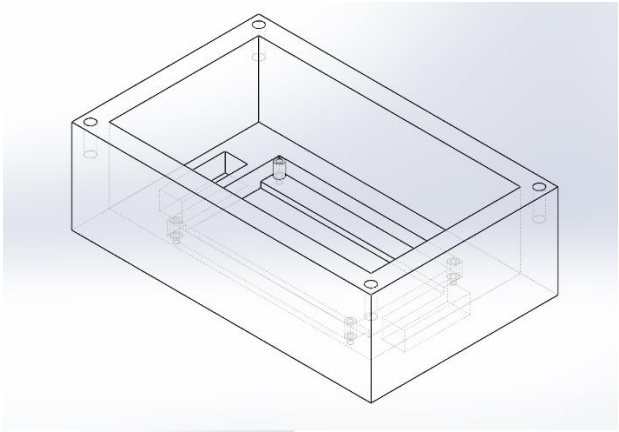
Procédure du Dobot gérant la construction de la ville

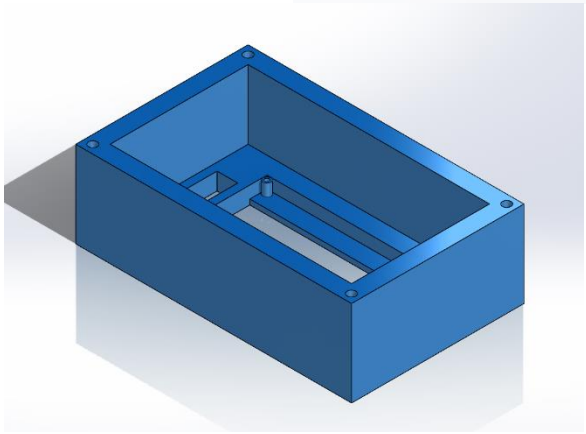
```
bool PerformBuildingStep(DobotBuilderUnit* bot)
{
   // Récupérer la structure en construction
   auto target = mapHandler.GetTargetSlot();
   if(target == nullptr)
   { // La ville est finie de construire
      perform_delay(WAIT_CAUSE_JOB_DONE);
      return false;
   }

   // Calcul des coordonnées dans le repère du Dobot
   float x, y, z;
   target->GetRealCurrentPosition(&x, &y, &z);
   compute_dobot_coordinates(1, &x, &y, &z);

   // Aller cherche le bloc
   set_dobot_working(1, true, false); // Indiquer le début de la procédure
   bot->SetGripping(false, true); // Ouvrir la pince
   bot->MoveTo(JUMP_XYZ, computedTransitionSlot[1][0],
computedTransitionSlot[1][1], offset[1][2] + UNIT_Z_SIZE - 5, 0); // Aller
sur la zone de transition
   bot->SetGripping(true, true); // Fermer la pince
   bot->SerialProcess(nullptr); // Envoyer les paquets

   if(y > 100) bot->MoveTo(MOVL_XYZ, computedTransitionSlot[1][0] + 50,
computedTransitionSlot[1][1], 2 * UNIT_Z_SIZE, 0); // Assurer que le robot
puisse atteindre toutes les coordonnées

   // Aller le placer
   bot->MoveTo(JUMP_XYZ, x, y, offset[1][2] + z + UNIT_Z_SIZE - 20, -target-
>rot + 90); // Se déplacer sur la structure en construction
   bot->SetGripping(false, true); // Ouvrir la pince
   bot->Delay(500, true); // Attendre 500 ms
   bot->SerialProcess(nullptr); // Envoyer les paquets

   bot->DisableGripper(true); // Eteindre la pompe
   // Retour en position initiale
   bot->MoveTo(JUMP_XYZ, 225, 0, 0, 0); // Retourner à la position initiale
   set_dobot_working(1, false, true); // Indiquer la fin de la procédure
   bot->SerialProcess(nullptr); // Envoyer les paquets

   mapHandler.ProgressConfirmed(); // Valider le progrès dans la construction
de la ville

   // Attendre que la première instruction soit traitée par le Dobot
   wait_until_working(1, true);
   // Réinitialiser le contenu de la zone de transition
   mapHandler.SetTransitionSlotContent(T_NONE);

   return true;
}
```

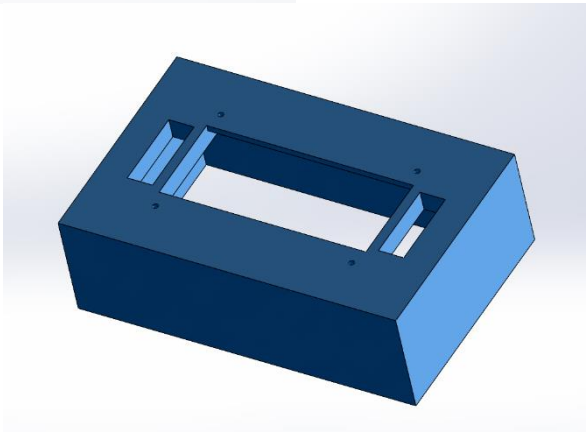
ANNEXE 8 : VUES D'UN BOITIER MODELISE SUR SOLIDWORKS

Les images suivantes montrent le boîtier que nous avons modélisé sous différents angles. Nous n'avons malheureusement pas pu l'imprimer par manque de temps.

Partie supérieure (boîtier)







Partie inférieure (couvercle)