Rapport du projet Robotique

Equipe HELMS

2023 - 2024

Enseignants :

- Nicolas BASKIOTIS
- Olivier SIGAUD

Étudiants :

- Sarah BEN MIMOUN
- Hugo Knecht (Absent)
- Micheal LI
- Lucie PAN
- Eric ZHENG

Table des matières

1 Introduction du Projet

Pour accéder au dépôt GitHub, cliquez ici.

Le projet que nous avons effectué ce semestre consistait à develloper un robot autonome capable d'effectuer des tâches basiques telles que : faire un carré, utiliser un capteur de distance et une caméra. Pour atteindre cet objectif, nous avons commencé par la création d'une simulation en python orientée objet. Cette simulation nous a permis de visualiser notamment grâce à `Tkinter` le comportement du robot dans un environnement virtuel, de tester différentes stratégies et de valider notre implémentation avant de passer à la phase de déploiement sur un véritable robot physique.

Nous avons appris dans un premier temps à utiliser gitHub pour pouvoir travailler en groupe sur le projet et nous nous sommes initier au langage python orientée objet.

Nous avons dans les premières semaines opté pour un retour sur le terminal et nous avons ensuite basculer pour un retour sur l'interface graphique `Tkinter`. Et enfin nous avons ajouté la 3D avec `Panda3D`.

Nous nous sommes répartis le travail grâce à `Trello` où nous pouvions créer des taches à effectuer pour la semaines et ainsi avoir une vu d'ensemble du travail à fournir.

2 Choix de conception

Dans le cadre de notre cours de Projet Robotique, nous avons entrepris une démarche ambitieuse visant à concevoir et développer un robot basé sur l'architecture Modèle-Vue-Contrôleur (MVC). Cette approche, largement utilisée dans le développement logiciel, offre un cadre structuré pour la conception de systèmes complexes en séparant les préoccupations liées aux données, à la logique de traitement et à l'interface utilisateur.

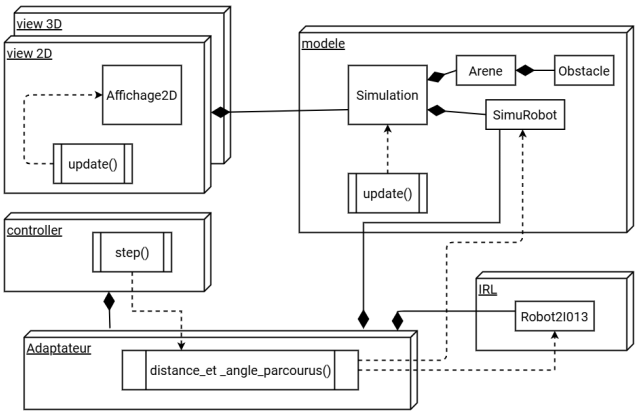


FIGURE 1 – État des lieux

Pour atteindre nos objectifs, nous avons utilisé plusieurs bibliothèques :

- L'affichage des messages : Logging permettant de garder un historique des événements et de l'état du systèm, et de détecter rapidement les anomalies.
- La simulation en 2d : Tkinter pour l'affichage pour sa simplicité d'utilisation et sa portabilité.
- La simulation en 3d : Panda3d pour l'affichage (simple à utiliser avec Python) et OpenCV pour l'enregistrement de image.
- La reconnaissance de la balise : Pillow et NumPy

Nous avons également utilisé des classes héritées pour structurer notre code de manière efficace et réutilisable, favorisant ainsi une meilleure modularité et maintenance du système.

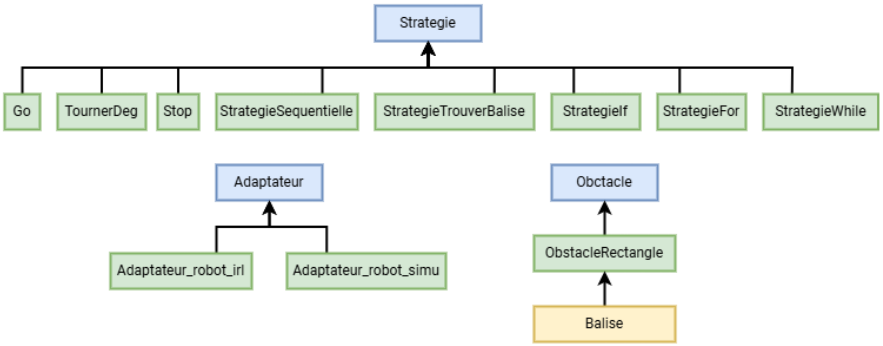


FIGURE 2 – Héritages existant dans le projet

3 Description du code

3.1 Architecture du code

Dans le dépôt GitHub, nous pouvons observer cinq dossiers principaux et plusieurs fichiers :

- Le fichier `main.py` permet d'exécuter la simulation du robot virtuel.
- Le fichier `main_robot_irl.py` permet d'exécuter la simulation du robot réel.
- Le dossier `strategies_prefaites` contient des stratégies déjà pré-écrites, comme demander au robot d'avancer ou de faire un carré.
- Le dossier `autre_documents` contient la preuve physique utilisée dans notre simulation et une fiche de renseignement sur les membres de l'équipe.
- Le dossier `videos_demo_strategie` contient les vidéos des tests réalisés sur le robot réel.
- Le dossier `docs` contient la docstring du projet en `html`.

3.2 Description des fichiers dans le `src`

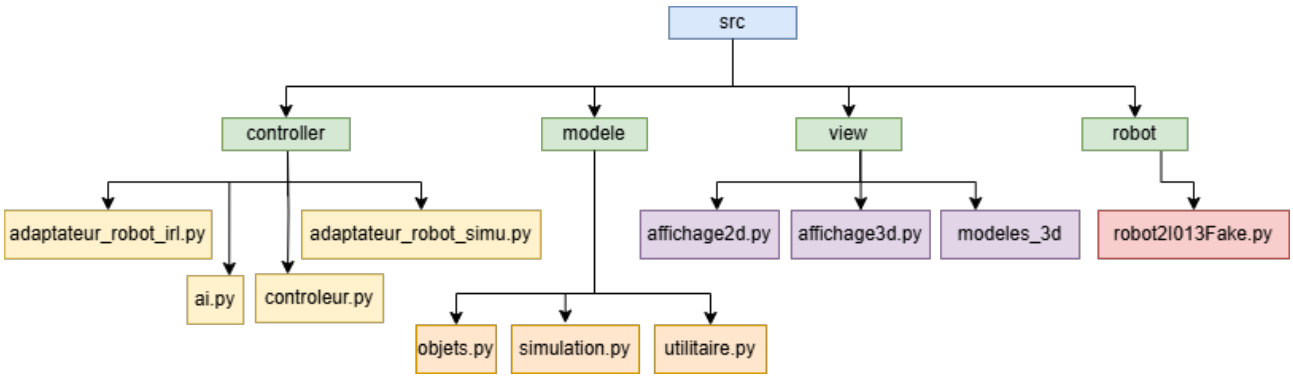


FIGURE 3 – Architecture du dossier `src`

Le dossier `src` est subdivisé en quatre modules : un module pour les préoccupations liées aux données (`modele`), un module pour la logique de traitement (`controller`), un module pour l'interface utilisateur (`view`) et un module pour l'API du robot (`robot`).

3.2.1 Le module `modele`

Le module `modele` possède un sous-module contenant toutes les classes d'objets (robot, obstacles, arène, etc.), un sous-module `utilitaires.py` contenant les fonctions mathématiques, et un sous-module `simulation.py` qui simule le robot.

3.2.2 Le module `controller`

Le module `controller` possède un sous-module `controleur.py`, un sous-module `ai.py` où se trouvent les stratégies de haut niveau et de bas niveau, un sous-module `adaptateur_robot_irl.py` (resp. `adaptateur_robot_simu.py`) qui fait l'intermédiaire entre le robot réel (resp. simulé) et le contrôleur.

3.2.3 Le module `view`

Le module `view` est composé d'un dossier `modeles_3d` et de deux sous-modules : `affichage2d.py` et `affichage3d.py`. Dans le dossier `modeles_3d`, vous trouverez les modèles 3D de la simulation (robot, obstacles, arène et balise) qui sont soit en `.glb` soit en `.egg`. Il y a également les patrons de ces objets dans `modeles_3d/blender`.

- Le sous-module `affichage2d.py` permet d'afficher en temps réel une simulation 2D à l'écran (cf. FIGURE 4). Les données du robot sont affichées en haut à gauche. Vous pouvez exécuter différentes stratégies en saisissant des valeurs pour les différentes variables. Certaines valeurs sont interdites et l'interface préviendra l'utilisateur par un message rouge. Il y a également un bouton permettant de remettre le robot à sa position initiale. Si la stratégie que vous souhaitez exécuter n'est pas présente, référez-vous à la section Application et test des stratégies

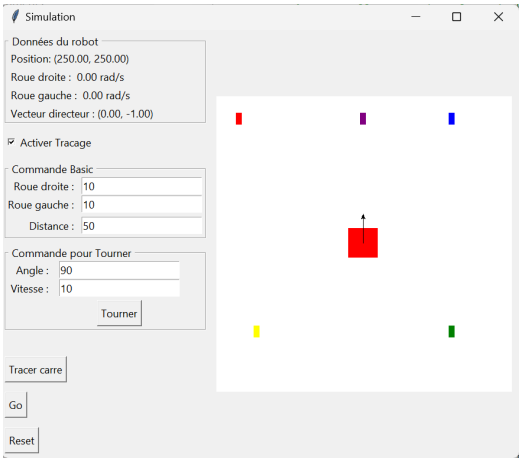
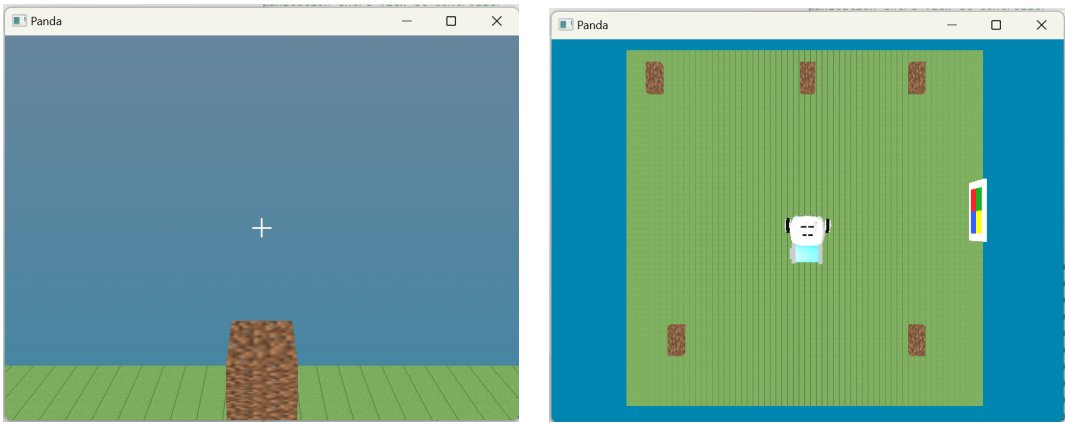


FIGURE 4 – Représentation en 2D du robot et de son environnement

- Le sous-module `affichage3d.py` permet d'afficher en temps réel une simulation 3D à l'écran (cf. FIGURE 5). Vous pouvez changer de point de vu (passer de la 3D en 2D) en appuyant sur la touche `esc` du clavier. Vous pouvez également pivoter (resp. incliner) la camera avec les touches `droite/gauche` (resp.`haut/bas`). Les stratégies déjà présentes

sur l'interface 2D peuvent csimuler le robot. Si la stratégie que vous souhaitez exécuter n'est pas présente, référez-vous à la section Application et test des stratégies



(a) Point de vue du robot (b) Vue de dessus (2D)

FIGURE 5 – Représentation en 3D du robot et de son environnement

3.2.4 Le module `robot`

Le module `robot` contient une API. Cette API est déployée lorsque l'API du robot réel n'a pas été trouvée lors d'une exécution.

4 Stratégie du robot

Nous avons développé plusieurs stratégies permettant de donner des ordres au robot réel ou simulé. Il existe des stratégies basiques et des stratégies imbriqués (des méga-stratégie). Chaque stratégie est une classe fille de la classe mère de `Strategie`. Leur structure sont identiques :

- `def __init__` permettant d'initialiser la stratégie.
- `def start` permettant de lancer la stratégie.
- `def stop` permettant de savoir quand elle est finie
- `def step` permettant de faire un pas dans la stratégie

Les stratégies sont exécutés par le `Controleur` avec la fonction `start`.
Pour créer sa propre stratégie et l'exécuter vous pouvez combiner les différentes classes de stratégie au niveau du `main` et utiliser la fonction `add_strat` du `Controleur` pour dire au robot d'effectuer la stratégie.

4.1 Strategie Basique

4.1.1 `Go`

Lorsque la stratégie `Go` est lancée avec les variables `adaptateur`, `distance`, `v_ang_d`, `v_ang_g`, et `active_trace`, l'`adaptateur` va mettre à jour le moteur gauche (resp. droit) du robot à `v_ang_d` (resp. `v_ang_g`) avec la fonction `set_vitesse_roue`. Puis, à chaque pas de temps, l'`adaptateur` vérifie si la `distance` à parcourir est atteinte avec (`distance_parcourue`). Par

ailleurs, si la variable `active_trace` est vraie, alors le robot tracera son parcours (cela ne fonctionnera qu'en 2D). (cf FIGURE 6.a)

4.1.2 `TourneDeg`

Lorsque la stratégie `TournerDeg` lancée avec les variables `adaptateur`, `angle`, `v_ang`, et `active_trace`, l'`adaptateur` va mettre à jour le moteur gauche et droit du robot à `v_ang`. Puis à chaque pas de temps l'`adaptateur` vérifie si l'`angle` à faire est atteint avec (`distance_parcourrue`). Idem pour le traçage. (cf FIGURE 6.b)

4.1.3 `Stop`

La stratégie `Stop` est une stratégie qui arrête le robot. Elle est utilisée pour arrêter le mouvement des roues du robot en mettant leur vitesse à zéro. Cette stratégie est toujours considérée comme terminée dès qu'elle commence. (cf FIGURE 6.c)

```python
class Go(Strategie):
    def start(self):
        """Commencer la strategie"""
        self.logger.info("Starting Go strategy")
        self.adaptateur.active_trace(self.active_trace)
        self.adaptateur.set_vitesse_roue(self.v_ang_d, self.v_ang_g)
        self.pos_ini = self.adaptateur.distance_parcourue()
        self.parcouru = 0

    def stop(self):
        """Verifier si la strategie est fini ou non
        :return: True si la strategie est finie, False sinon"""
        self.logger.info(f"distance parcourue {self.parcouru}, distance {self.di
        return math.fabs(self.parcouru) >= math.fabs(self.distance)

    def step(self):
        """pas de la strategie """
        self.parcouru += self.adaptateur.distance_parcourue()
        if self.stop():
            self.adaptateur.stop()
            self.logger.info("Go strategy finished")
            return
```

(a)Code de la stratégie `Go`

```python
class TournerDeg(Strategie):
    def start(self):
        """Commencer la strategie"""
        self.logger.info("Starting TournerDeg strategy")
        #self.logger.info(f"adapateur angle {self.adaptateur.angle}")
        self.adaptateur.active_trace(self.active_trace)
        if self.angle is None :
            if self.adaptateur.angle < 90:
                self.angle = 90 - self.adaptateur.angle
            else :
                self.angle = self.adaptateur.angle-90
        if self.angle > 0:
            self.v_ang_d, self.v_ang_g = self.v_ang, -self.v_ang
            self.adaptateur.set_vitesse_roue(self.v_ang, -self.v_ang)
        else:
            self.v_ang_d, self.v_ang_g = -self.v_ang, self.v_ang
            self.adaptateur.set_vitesse_roue(-self.v_ang, self.v_ang)
        self.pos_ini = 1. * self.adaptateur.angle_parcouru()
        self.parcouru = 0

    def stop(self) -> bool:
        """Verifier si la strategie est fini ou non
        :return: True si la strategie est finie ou False sinon"""
        self.logger.info(f"angle parcourue {self.parcouru}, angle {self.angle}")
        if self.angle == 0 or math.fabs(self.parcouru) >= math.fabs(self.angle):
            self.adaptateur.stop()
            return True
        return False

    def step(self):
        """pas de la strategie"""
        self.parcouru += self.adaptateur.angle_parcouru()
        if self.stop():
            self.logger.info("TournerDeg strategy finished")
            return
```

(b)Code de la stratégie `TournerDeg`

```python
class Stop(Strategie):
    def start(self):
        """commencer la strategie"""
        self.logger.info("Starting STOP strategy")
        self.adaptateur.set_vitesse_roue(0, 0)

    def stop(self) -> bool:
        """Verifier si la strategie est finie
        :return: True si la strategie est finie, False sinon"""
        return True

    def step(self):
        pass
```

(c)Code de la stratégie `Stop`

FIGURE 6 – Code des stratégies de base

4.2 Méta-stratégie

4.2.1 `StrategieSequentielle`

Cette approche permet de combiner plusieurs stratégies de base en une séquence, offrant ainsi au robot la possibilité de suivre une série d'actions (`steps`) prédéfinies de manière ordonnée. Cela rend la stratégie modulable et flexible, chaque action ou groupe d'actions étant en-capsulé dans des objets de stratégie distincts.

La fonction `stop` vérifie si toutes les étapes de la séquence ont été exécutées. Elle renvoie `True` si l'index de l'étape actuelle (`current_step`) est supérieur ou égal à la longueur de la liste des étapes (`len(self.steps)`), ce qui signifie que toutes les étapes ont été exécutées. `False` sinon.

La fonction `step` exécute un pas de la stratégie séquentielle. Si la stratégie est terminée (`self.stop()` retourne `True`), elle ne fait rien. Sinon, elle exécute un pas de l'étape actuelle (`self.steps[self.current_step].step()`). Si l'étape actuelle est terminée (`self.steps[self.current_step].stop()` retourne `True`), elle passe à l'étape suivante (`self.current_step += 1`) et démarre cette nouvelle étape si elle existe (`self.steps[self.current_step].start()`). (cf FIGURE 7)

```
122  class StrategieSequentielle(Strategie):
145      def stop(self) -> bool:
146          """Verifier si la strategie est finie
147          :return: True si la strategie est finie, False sinon"""
148          return self.current_step >= len(self.steps)
149
150      def step(self):
151          """pas de la strategie sequentielle """
152          if self.stop():
153              return
154          self.steps[self.current_step].step()
155          if self.steps[self.current_step].stop():
156              self.current_step += 1
157              if self.current_step < len(self.steps):
158                  self.steps[self.current_step].start()
```

FIGURE 7 – Code de la stratégie `StrategieSequentielle`

4.2.2 `StrategieIf`

La stratégie `StrategieIf` permet de conditionner l'exécution entre deux stratégies en fonction d'une condition spécifique, permettant une flexibilité dans le comportement du robot basé sur ses capteurs.

La fonction `start` initialise la stratégie conditionnelle en choisissant la stratégie à exécuter en fonction de la condition. Nous vérifions la distance mesurée par l'adaptateur (avec `get_distance()`) et nous comparons cette valeur avec la condition. Si la distance est supérieure ou égale à la condition, `stratA` est choisie. Sinon, `stratB` est choisie. (cf FIGURE 8)

```python
class StrategieIf(Strategie):
    def start(self):
        """commencer la strategie"""
        self.logger.info("Starting condition strategy")
        #self.logger.info(f"capteur distance {self.adaptateur.get_distance()}"
        if self.adaptateur.get_distance() >= self.condition:
            self.strat_a_faire = self.stratA
        else:
            self.strat_a_faire = self.stratB
        self.strat_a_faire.start()

    def stop(self) -> bool:
        """Verifier si la strategie est finie
        :return: True si la strategie est finie, False sinon"""
        # Verifier si strat_a_faire n'est pas vide
        if self.strat_a_faire is not None:
            return self.strat_a_faire.stop()

    def step(self):
        """pas de la strategie sequentielle """
        if self.strat_a_faire is not None:
            self.strat_a_faire.step()
```

FIGURE 8 – Code de la stratégie `StrategieIf`

4.2.3 `StrategieWhile`

La stratégie `StrategieWhile` permet de répéter une stratégie tant qu'une condition spécifique est vérifiée. Lorsque la condition n'est plus vérifiée, la stratégie s'arrête. Cette structure permet de gérer des boucles de comportement basées sur des conditions spécifiques, les mesures de capteurs.
Le `stop` vérifie si la condition est toujours vérifiée pour continuer la stratégie. Elle renvoie `True` si la distance mesurée par l'adaptateur (`get_distance()`) est inférieure à la condition. Nous arrêtons les moteurs du robot (`self.adaptateur.stop()`). Sinon si la stratégie `strat` est terminée (`self.strat.stop()` retourne `True`), elle est redémarrée (`self.strat.start()`). `False` sinon.
(Exemples d'application : la 2D et la 3D)

```python
class StrategieWhile(Strategie):
    def stop(self) -> bool:
        """Verifier si la strategie est finie
        :return: True si la strategie est finie, False sinon"""
        self.logger.info(f"capteur distance :  {self.adaptateur.get_distance()},
        if self.adaptateur.get_distance() < self.condition :
            self.logger.info(f"STOP !!! ")
            self.adaptateur.stop()
            return True
        else :
            if self.strat.stop():
                self.strat.start()
                return False

    def step(self):
        """pas de la strategie sequentielle """
        if self.stop():
            return
        self.strat.step()
```

FIGURE 9 – Code de la stratégie `StrategieWhile`

4.2.4 `StrategieFor`

La stratégie `StrategieFor` encapsule une stratégie et la répète plusieurs fois de manière séquentielle. Elle suit et contrôle le nombre de répétitions, assurant que la stratégie est exécutée le nombre de fois demandé avant de s'arrêter.

```python
class StrategieFor(Strategie):
    def start(self):
        """commencer la strategie"""
        self.logger.info("Starting For strategy")
        self.cpt = 0
        self.strategie.start()

    def stop(self) -> bool:
        """Verifier si la strategie est finie
        :return: True si la strategie est finie, False sinon"""
        if self.strategie.stop():
            self.cpt += 1
            if self.cpt >= self.nombre_repetition:
                return True
            self.strategie.start()
            return False

    def step(self):
        """pas de la strategie sequentielle """
        if self.stop():
            return
        self.strategie.step()
```

FIGURE 10 – Code de la stratégie `StrategieFor`

4.2.5 `StrategieTrouverBalise`

La stratégie `StrategieTrouverBalise` est une stratégie pour rechercher une balise en capturant et en analysant des images à différents angles. Elle utilise un `adaptateur` pour contrôler le robot et un `servo_moteur` pour ajuster l'angle de vision (`angle`). La stratégie s'arrête lorsqu'elle trouve la balise aide de la fonction `_process_image` ou lorsque le nombre maximal de captures est atteint.

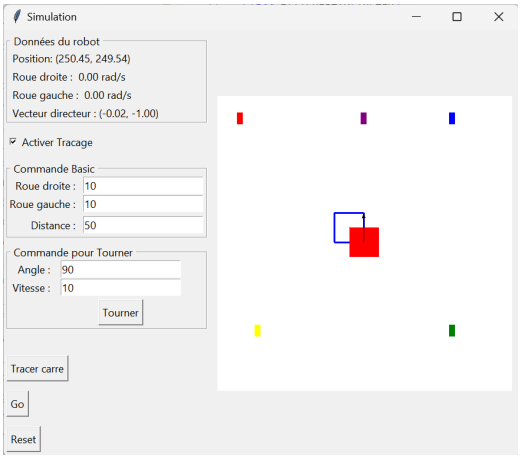
```python
class StrategieTrouverBalise(Strategie):
    def start(self):
        """Commencer la strategie"""
        self.logger.info("Starting StrategieTrouverBalise strategy")
        self.adaptateur.start_recording()
        self.adaptateur.servo_rotate(self.angle)

    def stop(self) -> bool:
        """Vérifier si la stratégie est finie
        :return: True si la stratégie est finie, False sinon"""
        if self._process_image():
            self.logger.info(f"Balise trouvée: {self.num}")
            return True
        self.logger.info(self.cpt)
        return self.cpt > self.seuil_cpt

    def step(self):
        """Pas de la stratégie"""
        if self.stop():
            self.adaptateur.stop_recording()
            return
        self.image = self.adaptateur.get_image()
        if self.image is not None:
            self.angle += self.pas_angle
            self.adaptateur.servo_rotate(self.angle)
            self.cpt += 1
```

FIGURE 11 – Code de la stratégie `StrategieTrouverBalise`

5 Application et test des stratégies

5.1 Objectif 1 : Tracer un carré

Le premier objectif de cette UE était de demander au robot de tracer un carré. Pour cela nous avons définit une stratégie séquentielle (`StrategieSequentielle`) qui avance puis tourne, et cette stratégie est répétée 4 de fois pour tracer un carré, (`StrategieFor`).

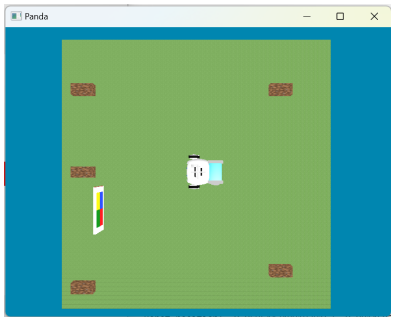


(a) Simulation de la stratégie en 2d

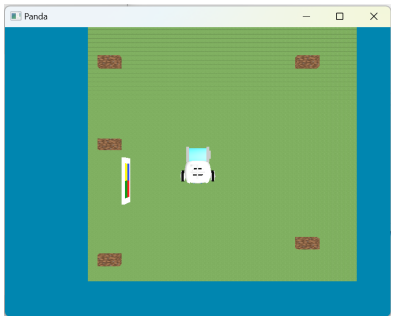
```python
if __name__ == '__main__':
    demo = Demo()
    #variable pour les strategies
    distance = 50
    vitesse = 10
    angle = 90
    activer_tracer = True
    nb_repetition = 4

    # strategies pour faire un carre
    avancer = Go(demo.adaptateur, distance, vitesse, vitesse, activer_tracer)
    touner = TournerDeg(demo.adaptateur, angle, vitesse, activer_tracer)
    strat_repeter = StrategieSequentielle(demo.adaptateur, [avancer, touner])
    tracer_carre = StrategieFor(demo.adaptateur, strat_repeter, nb_repetition)

    #ajout de la strategie
    demo.controller.add_strat(tracer_carre)
```

(b) Code permmettant d'effectuer la simulation

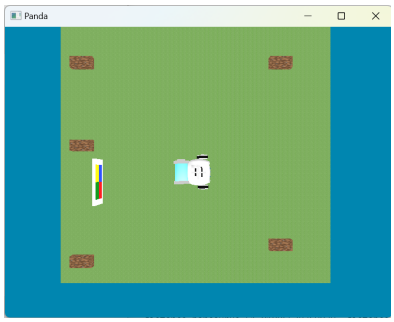
FIGURE 12 – Robot qui trace un carré



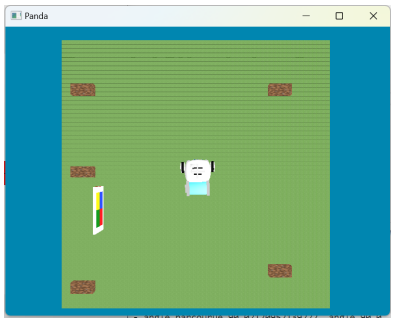
(a) Avant l'affectation de la stratégie t_0



(b) Position du robot à un $t_1 = t_0 + \Delta t$



(c) Position du robot à un $t_2 = t_1 + \Delta t$

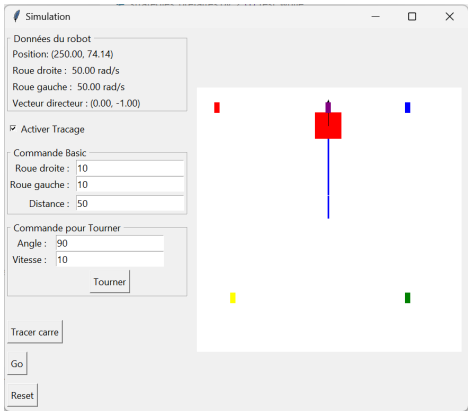


(d) Fin de la stratégie

FIGURE 13 – Évolution dans le temps lorsque robot effectue un carré

5.2 Objectif 2 : Foncer sur un mur/obstacle sans le toucher

Le deuxième objectif de cette UE était de demander au robot d'avancer vers un mur et de s'arreter juste avant de le toucher. Pour cela nous avons définit une stratégie (`StrategieWhile`) pour avancer jusqu'à ce qu'une collision soit détectée, où la détection de collision est déterminée par la condition `seuil_collision`.

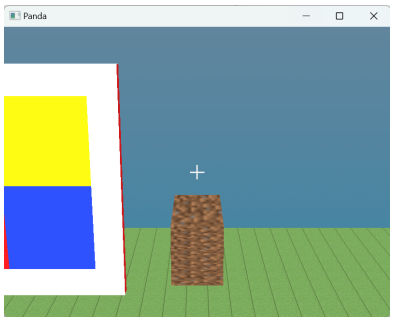


(a) Simulation de la stratégie en 2d

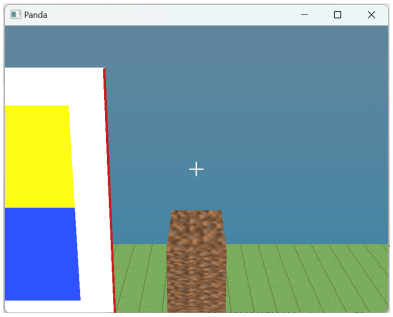
```
84  if __name__ == '__main__':
85      demo = Demo()
86      #variable pour les strategies
87      distance = 50
88      vitesse = 50
89      seuil_collision = 1
90
91      # strategies aller jusqu'au mur
92      avancer = Go(demo.controller.adaptateur, distance, vitesse, vitesse, True)
93      avancer_jusquau_mur = StrategieWhile(demo.adaptateur, avancer, seuil_collision)
94
95      #ajout de la strategie
96      demo.controller.add_strat(avancer_jusquau_mur)
```

(b) Code permmettant d'effectuer la simulation

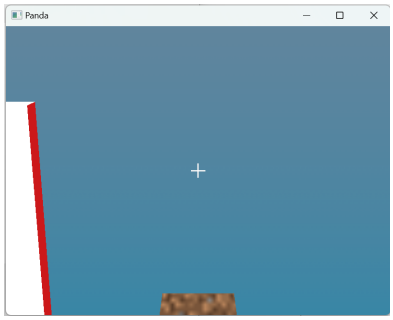
FIGURE 14 – Robot qui fonce sur un obstacle sans le toucher



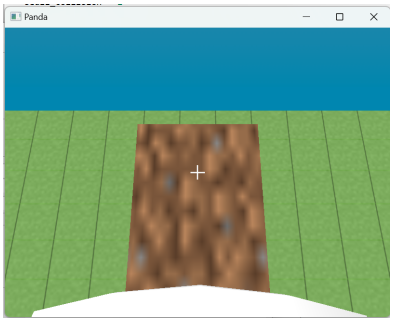
(a) Avant l'affectation de la stratégie t_0



(b) Position du robot à un $t_1 = t_0 + \Delta t$



(c) Position du robot à un $t_2 = t_1 + \Delta t$



(d) Fin de la stratégie, vue avec caméra qui a été inclinée

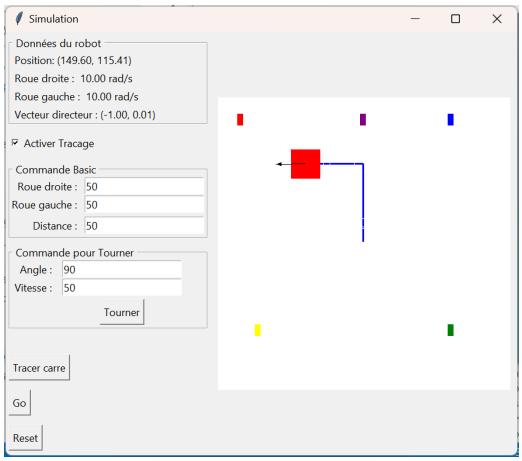
FIGURE 15 – Évolution dans le temps lorsque robot effectue une stratégie while

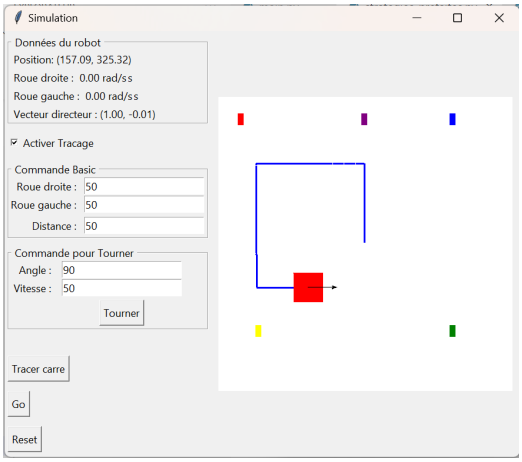
5.3 Objectif 3 : Reconnaissance de la balise

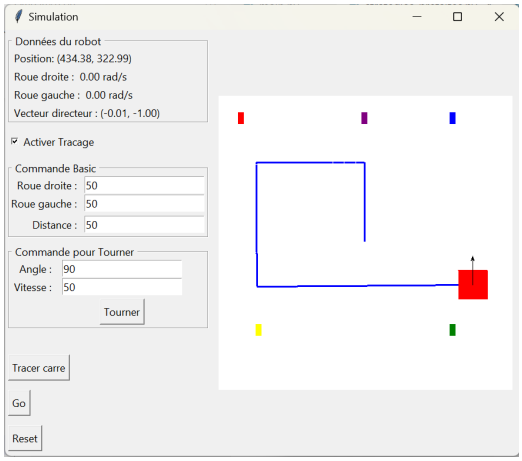
Le dernier objectif de cette UE était d'amener le robot à identifier une balise. À cette fin, nous avons élaboré une stratégie nommée `StrategieTrouverBalise`, qui analyse régulièrement le champ de vision du robot. À chaque intervalle défini (`pas_angle`), la stratégie vérifie la présence de la balise dans le champ de vision. Bien que cet objectif n'ait pas été accompli avec succès sur le robot simulé, il a été pleinement réalisé sur le robot réel. Vous pouvez trouver plus de détails dans la section Robot irl.

5.4 Autre : Éviter les obstacles

Nous avons décidé de définir un nouvel objectif : créer une stratégie qui guide le robot en ligne droite jusqu'à ce qu'il rencontre un obstacle, puis le tourne vers la gauche de 90 degrés. Ensuite, le robot reprend sa trajectoire en ligne droite jusqu'à ce qu'il rencontre un nouvel obstacle. Cette séquence d'actions doit être répétée quatre fois. Pour atteindre cet objectif, nous avons combiné les fonctionnalités des méta-stratégies `StrategieWhile` et `StrategieFor`, ainsi que les actions `Go` et `TournerDeg`.

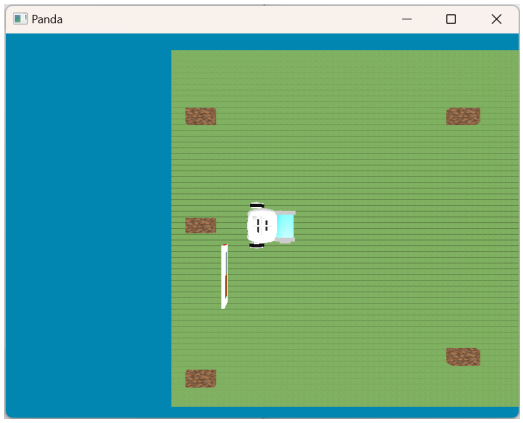




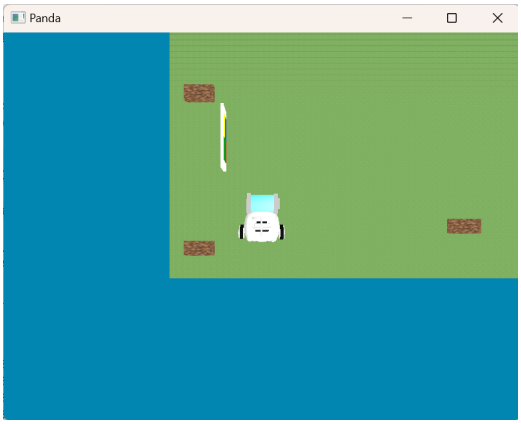



```python
if __name__ == '__main__':
    demo = Demo()
    #variable pour les strategies
    distance = 50
    vitesse = 10
    angle = 90
    activer_tracer = True
    nb_repetition = 4

    # strategies pour faire un carre
    avancer = Go(demo.adaptateur, distance, vitesse, vitesse, activer_tracer)
    touner = TournerDeg(demo.adaptateur, angle, vitesse, activer_tracer)
    strat_repeter = StrategieSequentielle(demo.adaptateur, [avancer, touner])
    tracer_carre = StrategieFor(demo.adaptateur, strat_repeter, nb_repetition)

    #ajout de la strategie
    demo.controller.add_strat(tracer_carre)
```

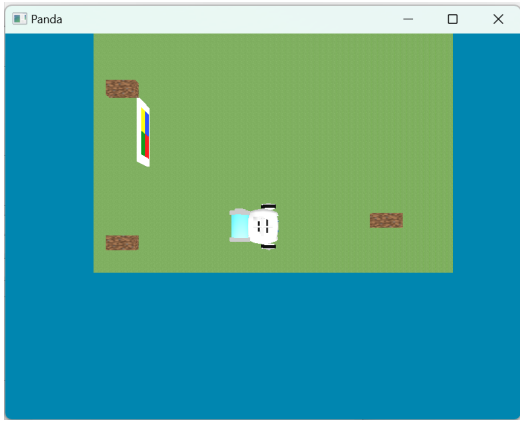
FIGURE 16 – Robot doit avancer vers l'obstacle puis tourne à gauche



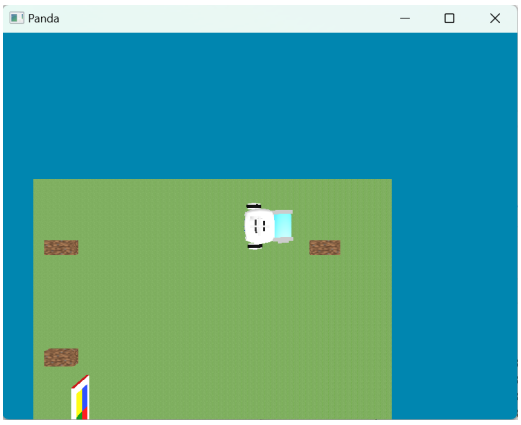
(a) Avant l'affectation de la stratégie t_0



(b) Position du robot à un $t_1 = t_0 + \Delta t$



(c) Position du robot à un $t_2 = t_1 + \Delta t$



(d) Fin de la stratégie, vue avec caméra a été inclinée

FIGURE 17 – Évolution dans le temps lorsque robot effectue une stratégie while

6 Robot IRL

Le robot réel utilisé dans notre projet est possède :

- Un Raspberry Pi
- Une carte contrôleur (Arduino)
- Deux moteurs encodeurs pour le contrôle des roues
- 3 senseurs : une caméra, un capteur de distance, un accéléromètre

Nous l'avons utilisé pour tester les stratégies que nous avons programmé. Vous trouverez ci-dessous les vidéos :

Pour visionner le test de rotation : le robot tourne à 90 degrés vers la droite. Cliquez ici.

Pour visionner le test de stratégie conditionnelle :

- Si le robot est à plus de 100 mm, il ne fait rien ; sinon, il avance de 20 mm. Cliquez ici.
- Idem mais en inversant les conditions. Cliquez ici.

Pour visionner le test d'une stratégie while : tant que le robot est à plus de 50 mm d'un obstacle, il avance de 20 mm.

- Cas 1 : le robot est à 150 mm de son obstacle. Cliquez ici.
- Cas 2 : le robot est à moins de 50 mm de son obstacle. Cliquez ici.

Pour visionner le test de la réalisation d'un carrée de 150 mm. Cliquez ici.

Pour visionner le test de reconnaissance de la balise : le robot regarde de la droite vers la gauche, commençant à 20° et pivotant à chaque fois de 10°. Cliquez ici.

Pour visionner le test trouver la balise et d'arrêt près d'elle, cliquez ici.

7 Difficultés rencontrés

Nous avons été confrontés à plusieurs défis tout au long des semaines de travail.

Dans un premier temps, nous avons dû nous familiariser avec l'utilisation de Trello. Cela nous a demandé un certain temps pour pouvoir estimer correctement la durée de chaque tâche et organiser leur répartition de manière à éviter la présence de sous-tâches inutiles.

Ensuite, nous avons rencontré des difficultés de factorisation de notre code. Les classes que nous avions implémentées étaient trop imbriquées les unes dans les autres, ce qui compliquait la gestion du programme si nous souhaitions désactiver certaines fonctionnalités.

La perte de membres de l'équipe en cours de projet peut représenter un défi significatif pour le reste de l'équipe. Lorsqu'un membre quitte le groupe, cela peut déséquilibrer la répartition des tâches et entraîner une perte de connaissances spécifiques qu'il ou elle avait apportées au projet. Malheureusement, c'est exactement ce qui s'est produit dans notre cas.

De plus, la répartition du temps de travail sur le projet variait selon les membres du groupe. Certains avaient d'autres priorités en dehors du projet, ce qui influençait le temps qu'ils pouvaient y consacrer. Par conséquent, vers la fin, il ne restait plus beaucoup de membres impliqués dans le projet, ce qui a eu un impact sur la progression et l'achèvement des tâches.

8 Conclusion

Le projet robotique a constitué une expérience riche en enseignements. Malgré les divers défis rencontrés, nous avons réussi à en surmonter la plupart. Bien que le projet ne soit pas entièrement achevé, en particulier la reconnaissance de la balise en 3D, nous avons néanmoins acquis des compétences et des connaissances précieuses. Cette aventure nous a également offert une exploration approfondie du domaine de la robotique et de la conception de logiciels.