Ré sumé des fonctions du projet se trouvant dans envs.py, _init_.py, definition.py (env prof)

dans le répertoire src/agents/team2 , on peut trouver notre agent 2:
initialement, dans

agent2.py:

Notre objectif est de passer d'un agent qui fait n'importe quoi (random) à un agent capable de suivre la piste et de gagner.
dans la fonction
pystk_process.py:
dans la fonction je vais expliquer les from et import:

1- from functools import partial, partialmethod:

funtools est un module standard de python qui sert à manipuler des fonctions.
partial: c'est une **fonction** définie dans functools qui sert à créer une nouvelle fonction à partir d'une autre, en **fixant certains arguments**.
import logging:
Un **module standard Python** pour écrire des messages de log(permet de decrire ce que fait le programme)

2- from multiprocessing import Pipe, Process:

multiprocessing:
Un module standard Python pour faire du **multi-process**.

Process
Une **classe** qui permet de créer un nouveau process Python. il sert à lancer une fonction dans un autre processus

Pipe : une fonction qui permet la communication du processus pere au processus fils

connection: c la classe des objets renvoyés par Pipe()

sys: donne acces au systeme python

3- from typing import List, Optional

Optional: le type est soit ce type soit None ex: Optional[int]
pystk2: pystk2 est une bibliothèque Python qui permet de contrôler le jeu SuperTuxKart depuis Python. SuperTuxKart est une machine de c++ donc on a besoin de traducteur.

on a la classe **class PySTKRemoteProcess**:
world; track; race sont les attributs.
pygtk2.Race = la course en cours (la simulation)

pygtk2.World State = l'état du monde (positions, phase, etc.)

pygtk2.Track = la piste (géométrie, segments)

__init__ *: C'est le* ***constructeur*** *: appelé quand on fait PySTKRemoteProcess(with_graphics).*

je ne vais pas detailler. c plus je crois pour l'affichage.

la méthode **list_tracks**: renvoie la liste des pistes disponibles dans STK
def list_tracks(self) -> List[str]:
La méthode **close(self)** permet d'arrêter proprement la course . self doit être remplacé par stk quand on l'utilise
warmup_race(config) :démarre une nouvelle course et attends que le STK soit prêt
il stoppe l'ancienne course si elle existe et crée une nouvelle course
get_world() : récupérer l'état actuel renvoie un message si pas de course sinon il met à jour les positions, vitesses, .. et renvoie World
race_step(*args) : avancer la simulation avec actions
get_kart_action(kart_ix) : lire l'action décidée par l'IA STK à utiliser que quand use_ai=True

II- envs.py:
1- les importations;
de typing, on a importé les types, **Any**, **Classvar**, **Dict**, **List**, **Optional**, **Tuple**, **TypedDict**
2- les fonctions:
def kart_action_space(): ne prend aucun paramètre. elle renvoie un dictionnaire de commandes possibles(les descriptions des commandes)

- ce que contient ce dictionnaire:

```
    # Acceleration
    "acceleration": spaces.Box(0, 1, shape=(1,)),
    # Steering angle
    "steer": spaces.Box(-1, 1, shape=(1,)),
    # Brake
    "brake": spaces.Discrete(2),
    # Drift
    "drift": spaces.Discrete(2),
    # Fire
    "fire": spaces.Discrete(2),
    # Nitro
    "nitro": spaces.Discrete(2),
    # Call the rescue bird
    "rescue": spaces.Discrete(2),
```

on peut accélérer entre 0 et 1: 0 ne pas accélerer 1 accélérer à fond.
steer: on peut tourner fort à gauche -1; tout droit 0; +1 tourner fort à droite
def kart_observation_space(use_ai): elle définit ce que l'agent reçoit dans obs(obs a été defini dans getstate; elle recoit space return par kart_obser…
obs["center_path_distance"]: C'est un **nombre** qui indique **à quelle distance le kart est du centre de la piste**.

- valeur proche de **0** → le kart est bien centré
- valeur **positive** → le kart est décalé d'un côté
- valeur **négative** → le kart est décalé de l'autre côté

obs["paths_start"] et obs["paths_end"] sont des **segments de piste devant le kart**.

Chaque segment est défini par :

- un point de départ (paths_start)
- un point d'arrivée (paths_end)

Les coordonnées sont données **dans le repère du kart** :

- **z** = devant
- **x** = gauche/droite2.

obs["velocity"]: C'est un vecteur de vitesse du kart.

- sa norme = vitesse
- sa direction = direction du mouvement

class BaseSTKRaceEnv(gym.Env[Any, STKAction]):

BaseSTKRaceEnv est la **classe de base** des environnements de course SuperTuxKart. Elle implémente toute la logique commune pour :

1. **initialiser** SuperTuxKart via un processus séparé (PySTKProcess),
2. **configurer** une course (reset_race),
3. **mettre à jour** l'état du monde (world_update),
4. **construire** l'observation (get_observation),
5. **calculer** reward/terminaison/infos (get_state).

LES MÉTHODES

1. **def initialize(self, with_graphics: bool):** **avoir STK prêt** et connaître les pistes.

   Crée _process si ce n'est pas déjà fait.
   Charge TRACKS en appelant list_tracks() dans le processus STK.

2. **def __init__(self, *, render_mode=None, track=None, num_kart=3, max_paths=None, laps: int = 1,difficulty: int = 2,):** **objectif :** préparer l'environnement, sans encore démarrer une course.
3. **def reset_race(random, options=None):** créer la configuration STK de la course à lancer

nb: il y a d'autres methodes dans la classe mais elles semblent se concentre beaucoup plus sur la course que sur notre agent. Donc, on ne va pas parler d'elles

4. **get_observation(kart_ix, use_ai):** Cette méthode transforme les données STK brutes en une observation exploitable :

   Cette méthode transforme les données STK brutes en une observation exploitable :

   a) Passage au repère du kart : kartview(x)

- transforme les coordonnées du monde en coordonnées relatives au kart.
- dans ce repère :
  - z = devant
  - x = gauche/droite

objectif : simplifier le pilotage (tout est vu "depuis le kart").

b) Trier items et karts "par pertinence"

- calcule positions relatives des autres karts et des items
- les trie en mettant d'abord ce qui est **devant** le kart

objectif: observation plus utile (évite un ordre aléatoire).

c) Calculer la distance au centre de la piste

- récupère le segment de piste courant (node kart.node)
- calcule un vecteur orthogonal vers la ligne centrale (x_orth)
- en déduit :
  - center_path_distance (distance signée)

- center_path (vecteur vers le centre)

d) Retour final

Renvoie un dictionnaire contenant notamment :

- velocity, max_steer_angle, distance_down_track
- center_path_distance, center_path
- items_position/items_type, karts_position
- paths_start/paths_end/paths_width/paths_distance

class STKRaceEnv(BaseSTKRaceEnv):

Dans STKRaceMultiEnv, l'environnement gère une course **multi-agent** : au lieu d'un seul obs, il renvoie **un dictionnaire d'observations** { "0": obs0, "1": obs1, ... }, et **chaque agent reçoit son propre obs** construit via get_observation. Les espaces Gym sont aussi des dictionnaires : action_space["0"] correspond à l'action de l'agent 0 et observation_space["0"] à son observation, ce qui signifie que chaque agent agit indépendamment tout en partageant la même course. Lors du reset(), chaque agent est associé à un kart précis, soit via rank_start (position imposée), soit via une position libre choisie automatiquement. L'environnement mémorise ensuite cette correspondance dans self.kart_indices, ce qui permet d'appliquer les actions au bon kart et de renvoyer l'observation correcte pour chaque agent.

semaine 3:

pystk_process.py:

a pour objectif d'exécuter SuperTuxKart (**pystk2**) dans **un processus séparé** et de fournir une interface simple au reste du projet (notamment envs.py). . Cette séparation permet ainsi d'isoler le moteur du jeu du code Gymnasium, et d'éviter que l'environnement principal ne se bloque ou ne plante.

I- La classe PySTKRemoteProcess

La classe **PySTKRemoteProcess** : s'exécute dans le **processus enfant** (le "serveur"). Il contient l'état du jeu et exécute réellement les commandes STK.

Initialisation (__init__)

Le constructeur initialise pystk2 avec ou sans graphismes :

- si with_graphics=True → GraphicsConfig.hd() (affichage)
- sinon → GraphicsConfig.none() (mode headless, plus rapide)

Boucle principale (run)

run() est la fonction exécutée dans le processus enfant. Elle :

1. configure le niveau de logs,
2. crée une instance stk,
3. boucle indéfiniment en attendant des commandes via pipe.recv().

Chaque commande reçue est un objet de type partialmethod représentant "quelle méthode appeler + quels arguments". La méthode est exécutée sur l'objet stk, puis le résultat est renvoyé au processus parent via pipe.send(result). Si la commande reçue est None, le processus s'arrête proprement.

Fonctions exposées

- list_tracks() : renvoie la liste des pistes disponibles.
- warmup_race(config) : démarre une course STK (création Race, start, puis plusieurs step() jusqu'à la phase READY) et renvoie l'objet Track. Cela garantit que le jeu est prêt avant de commencer les observations.
- get_world() : met à jour et renvoie l'état du monde (WorldState).
- race_step(*args) : avance la simulation d'un pas (avec ou sans actions selon le cas).
- get_kart_action(kart_ix) : récupère l'action choisie par l'IA STK pour un kart (utile quand use_ai=True).

II- PySTKProcess(processus parent)

La classe **PySTKProcess** : reste dans le **processus parent** (le "client"). Il envoie des commandes au processus enfant et récupère les résultats via un **Pipe**.

Création du processus

Le constructeur :

1. crée un Pipe (communication bidirectionnelle parent(<->) enfant),
2. démarre un Process Python dont la cible est PySTKRemoteProcess.run,
3. passe au processus enfant : with_graphics, le niveau de logs, et l'extrémité "remote" du pipe.

Exécution d'une commande (_run)

_run() envoie une commande au processus enfant et attend la réponse :

- il encapsule une méthode (ex: PySTKRemoteProcess.get_world) et ses arguments dans un partialmethod,
- l'envoie via self.pipe.send(...),
- récupère le résultat via self.pipe.recv().

Si le résultat est une Exception, elle est relancée côté parent (raise), ce qui remonte proprement les erreurs au code appelant.

stk_wrappers.py:

Le fichier stk_wrappers.py contient des **wrappers Gymnasium spécifiques à SuperTuxKart**.
Un wrapper est une couche autour de l'environnement qui **transforme** soit :

- les **observations** (ce que voit l'agent), soit
- les **actions** (ce que l'agent envoie),

sans modifier l'environnement de base

I- la classe PolarObservations

La classe PolarObservations est un wrapper d'observation qui transforme certaines positions 3D (x, y, z) dans le repère du kart en un triplet :

- **angle_zx** : angle calculé avec atan2(x, z), représentant la position gauche/droite par rapport à l'avant du kart
- **angle_zy** : angle calculé avec atan2(y, z), représentant la position haut/bas

- **distance** : norme du vecteur (x, y, z)

Ces trois valeurs remplacent directement les coordonnées d'origine.

La transformation est appliquée aux clés :
items_position, karts_position, paths_start, paths_end et center_path.

L'observation retournée conserve les mêmes clés, mais les vecteurs 3D sont remplacés par (angle_zx, angle_zy, distance). Les angles sont en radian

II- la classe ConstantSizedObservations

ConstantSizedObservations est un wrapper d'observation qui force certaines observations à avoir **une taille fixe**, indépendamment du nombre réel d'éléments présents dans la scène.

Il transforme les clés suivantes en tableaux de taille constante :

- paths_distance
- paths_width
- paths_start
- paths_end
- items_position
- items_type
- karts_position

Les tailles sont définies par :

- state_paths

- state_items
- state_karts

Si le nombre d'éléments est inférieur à la taille cible, le wrapper complète avec des valeurs par défaut. S'il est supérieur, il tronque les données.

Si add_mask=True, il ajoute :

- paths_mask
- items_mask
- karts_mask

Ces masques indiquent quels éléments sont réels (1) et lesquels sont du remplissage (0).

L'observation retournée conserve les mêmes clés, mais avec des tableaux de dimensions fixes.

III- DiscreteActionsWrapper

DiscreteActionsWrapper est un wrapper d'action qui remplace les actions continues acceleration et steer par des actions discrètes. L'action acceleration, initialement définie dans l'intervalle [0, 1], est discrétisée en un nombre fini de valeurs défini par acceleration_steps, et l'action steer, initialement définie dans [-1, 1], est discrétisée en un nombre fini de valeurs défini par steer_steps. L'agent fournit donc des indices discrets pour ces deux actions. Avant l'exécution dans l'environnement, le wrapper convertit automatiquement ces indices en valeurs continues correspondantes. Les autres actions (brake, drift, nitro, rescue, fire) ne sont pas modifiées. Si l'observation contient une clé action, celle-ci est également convertie en version discrète.

IV-OnlyContinuousActionsWrapper

Supprime les actions discrètes (booléennes) et ne garde que :

- accélération
- steer

Les actions supprimées (frein, drift, nitro, rescue, fire) sont forcées à 0 lors de l'envoi à l'environnement.

utils.py:

Le fichier utils.py contient des fonctions et classes utilitaires utilisées par l'environnement et les wrappers pour les calculs géométriques et la gestion des espaces discrets.

La fonction rotate(v, q) applique une rotation à un vecteur 3D v à l'aide d'un quaternion q. Le quaternion représente l'orientation du kart dans l'espace. Cette fonction permet de transformer des positions exprimées dans le repère du monde en positions exprimées dans le repère du kart (gauche/droite, haut/bas, devant/derrière). Elle est utilisée lors de la construction des observations pour fournir des informations relatives au kart plutôt qu'au monde global. La fonction est optimisée avec numba lorsque cette bibliothèque est disponible.

La fonction max_enum_value(EnumType) retourne la valeur maximale contenue dans une énumération, augmentée de un. Elle est utilisée pour déterminer automatiquement la taille des espaces discrets (Discrete et MultiDiscrete) à partir des énumérations définies dans pystk2 (items, power-ups, attachements, etc.).

La classe Discretizer permet de convertir une valeur continue appartenant à un intervalle défini par un spaces.Box en une valeur discrète, et inversement. Elle fournit une méthode discretize pour transformer une valeur continue en

indice discret, et une méthode continuous pour reconstruire une valeur continue à partir de cet indice. Cette classe est utilisée notamment pour la discrétisation des actions continues comme l'accélération et la direction dans les wrappers d'actions