COMPTE RENDU SEMAINE DU 02/01/2026:

fonction _calcul_angle(self,obs):
Notre fonction permet de calculer l'angle entre la trajectoire du nœud (milieu de segment avec path_start et path_end) vers le kart et la trajectoire du kart.

```python
def choose_action(self, obs):
    angle = self._calcul_angle(obs)
    print(f"Angle actuel: {angle:.3f} rad, {np.degrees(angle):.1f}°")

    acceleration = 0
    steering = -1.0
    action = {
        "acceleration": acceleration,
        "steer":  False, #np.clip(angle, -0.5, 0.5),
        "brake": False, #abs(angle) > 1.0, # bool(random.getrandbits(1)),
        "drift": False,
        "nitro": False,
        "rescue": False,
        "fire": False,
    }
    return action
```

Pour afficher les valeurs de l'angle a chaque frame (une superposition d'images affichées à l'écran). Ainsi à chaque frame, on obtient un ensemble d'angle qui précise dans quel coté nous sommes , par exemple si nous sommes à gauche l'angle en radian sera ainsi négatif et dans le cas où le kart est à droite , les angles en radian seront positives.

Algorithme suivi:
[#pour avoir le nb de noeud on fait len(obs.path_start)]

#vérifier si on est au début et qu'il y a bien y a des noeuds disponibles (si env existe)(obs ou self.env.track)
#si vérifié alors prend des vecteurs de positions depuis obs voir en bas (obs ou self.env.world.karts[idx]])

#obs = ("center_path", "path_nodes", "paths_start", "paths_end")
#chercher indice sécurisé (position d' un nœud qui existe vrm !)

#quand trouvé commencer le calcul d'angle calcule dx = node.x - kart.x et dz = node.z - kart.z, puis angle = atan2(dx, dz).
#node = nodes[idx][:3]
#dx = node[0] - pos[0]
#dz = node[2] - pos[2]
#ang_rad = np.arctan2(dx, dz) # convention : 0 = devant (z), >0 = à droite
#a chaque noeud atteint on reset le tableau des obs (pour ne pas avoir les mêmes observations des noeuds de la position précédente)

[#boucle possible , reset pour optimisée)]

Dans un premier temps nous avions suivi, cet algorithme cependant nous nous sommes rendus compte qu'il n'y a avait pas de nœuds. Ainsi depuis les segments (ce sont des paires de nœuds) de la trajectoire, nous avons donc calculé les coordonnées. La variable "cible" permet de viser le milieu de chaque segment afin que la direction change progressivement et ainsi avoir moins de zigzag, ainsi avoir une conduite du kart plus fluide. Cependant, nous avons remarqué que notre fonction d'angle renvoyait des valeurs aléatoires qui ne suivaient pas réellement la trajectoire de notre kart, nous avions par exemple un angle qui peut aller du positif vers le négatif même si notre kart est à la droite de la piste.

À l'affichage du terminal :
Angle actuel: 0.776 rad, 44.4°
Angle actuel: 1.338 rad, 76.6°
Angle actuel: 1.432 rad, 82.1°
Angle actuel: 1.264 rad, 72.4°
Angle actuel: 1.041 rad, 59.7°
Angle actuel: -2.644 rad, -151.5° ← incohérent avec les observations qu'on attendait
Angle actuel: 0.550 rad, 31.5°
.
.
.
.
Angle actuel: 1.384 rad, 79.3°

On se donne donc comme objectif de régler ce problème pour la semaine suivante. Et nous discuterons des éventuelles améliorations.

var en rad entre 6 et 0 (ou en degré) essayer de faire en sorte que la trajectoire du kart soit au milieu de la piste .
Comment marche le python objet. ←
préciser les commit et les modifications
faire une branche pour merger dans le main → une branche dev

Améliorations possibles:

- L'attribut "lookahead" pour choisir un nombre de nœuds visible pour être les plus strategique et rapide en fonction de la situation (obstacle, virage, saut..) rencontrés de l'observation du kart.

Réflexion faite:

- Plus il y a de nœud visible plus la conduite de l'agent sera fluide (mais peut être problème de vitesse -> la vitesse sera minimale) et le calcule de distance (= nb de nœuds) => choix de direction plus rapide, en cas de piste avec plusieurs chemins possible.

début d’une implémentation d’un code qui réagit à un obstacle:

```
#if target_item_distance== 10:
            if target_item_type in bad_type:
                if target_item_angle >5: # obstacle a droite
                    action["steer"]= - 0.5 # on tourne a gauche
                    action["nitro"]=False
                    action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner
                elif target_item_angle ==0 : # obstacle a droite
                    action["steer"]= - 0.5 # on tourne a gauche ou droite
                    action["nitro"]=False
                    action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner
                elif target_item_angle <-5: # obstacle a gauche
                    action["steer"]= 0.5 # on tourne a droite
                    action["nitro"]=False
                    action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner

            elif target_item_type in good_type:
                if target_item_angle <-5: # a gauche
                    action["steer"]= - 0.5 # on tourne a gauche
                    action["nitro"]=True
                    action["acceleration"]= action["acceleration"] + 0.5 #val a
determiner
                elif target_item_angle >5: #a droite
                    action["steer"]= 0.5 # on tourne a droite
                    action["nitro"]=True
                    action["acceleration"]= action["acceleration"] + 0.5 #val a
determiner


(ajout d'un code qui reagis aux obstacles dans choose_action)
        return action
```

Raisonnement :

La fonction choose action reçoit des observations (en provenance de l'environnement) et adapte sa réaction pour réagir aux obstacles de manière réfléchie.

Comment sont classés les obstacles et quels sont-ils dans ce jeu?

Les obstacles (ou item) sont au nombre de 10 et sont classés dans des listes selon s'ils sont bénéfiques à l'agent ou non. En effet, dans le fichier ItemObservationWrappers, les bons items sont rangés dans la liste good_types[ ] (ex: bonus_nitro) et les mauvais dans bad_types[ ] (ex: banane).

```
good_types = [0, 2, 3, 6]
bad_types = [1, 4]
# Assume a maximum of 10 different item types.
```

Quand est- ce qu'on prend en compte un item ?
Lorsqu'il est à une distance <10.

```
valid_mask = ahead_mask & (distances < 10) & (np.abs(angles) < 30)
```

Comment faire face à un bon et un mauvais temps?
Il faut éviter les mauvais et aller dans la direction des bons.

Nous avons à disposition des keys : target_item_position, target_item_distance, target_item_angle, target_item_type. Celles-ci sont utiles dans l'implémentation de la réaction de l'agent. Cependant, lorsqu'on lance la partie avec les modifications ci-dessus, aucun changement n'est constaté sur l'agent. Ainsi on peut supposer que le problème peut provenir d'une mauvaise utilisation des keys.

Les valeurs utilisées dans les if seront a adapté lorsque les keys auront été correctement utilisés. Ainsi nous pourrons tester différentes valeurs en fonction de nos observations du comportement de l'agent durant les courses.

Questions encore en suspens:

Est- ce qu'il faut ralentir et tourner face à un mauvais obstacle ou seulement tourner?
Comment modifier la vitesse du kart? avec la variable velocity? la clé "accélération"?
Comment utiliser Kart_obs_space dans l'implémentation de choose_action()? Est-ce nécessaire?

Leon:

1. Lorsqu'on vise le path qui se situe trop près de soi, elle provoque une **oscillation** de kart.
2. **Norme de vitesse** : Calcul de la scalaire du vecteur vitesse v = (vx, vy, vz). Il s'agit de la « vitesse absolue » indépendamment de la direction.
   2.1. speed = np.linalg.norm(velocity)
3. **obs["paths_start"]** : Coordonnées relatives des nœuds basées sur la position et l'orientation actuelles du kart. Ce n'est pas un système de coordonnées mondiales absolues, mais un système de coordonnées locales où le kart est toujours au centre (0, 0, 0) et l'avant correspond à l'axe z positif.
4. Type de donnée **velocity** : Un tableau de nombres Float de taille 3.
5. **stuck_steps** : Un dispositif qui surveille si le kart est bloqué et déclenche une recule.
6. **np.array** : La conversion en np.array permet d'effectuer des opérations mathématiques complexes (addition de vecteurs, soustraction, produit scalaire, calcul de norme).
7. **Phase**
   7.1. READY_PHASE
   7.2. SET_PHASE
   7.3. GO_PHASE : Juste après l'affichage de "Go" et le début de la course.
   7.4. RACE_PHASE : Course normale en cours.
8. **paths_start** : Liste des nœuds situés au centre de la piste.
9. shuffle : Les voitures sont fixées sur la piste, et **np.random.shuffle(agents)** simule un tirage au sort juste avant la course pour déterminer quel pilote monte dans quelle voiture.
10. **np.clip** : Limite les valeurs pour qu'elles ne dépassent pas l'intervalle [-1, 1].
11. Si l'on continue de freiner alors que le véhicule est à l'arrêt, alors elle passe en marche arrière.

Sokhna: