Mid Semester Report

Authored by: Wilson Cedric GENEVIEVE
Safa GUNES
Mahmoud DARWISH
Badr BENHAMMOU

Bachelor in Computer Science, Sorbonne University, Paris, France
January 2025 - May 2025

Supervised by: Olivier SIGAUD
Computer Science Professor and Machine Learning Researcher
ISIR Robotics Laboratory, Sorbonne University

Abstract. This project focuses on developing autonomous driving agents for *Super Tux Kart*, a popular open-source racing game. The primary objective is to create agents capable of navigating complex racing tracks efficiently while optimizing performance. The project involves implementing various agents, such as the `MedianAgent`, `EulerAgent`, and the recently introduced `ItemsAgent`, each designed to address distinct challenges within the game environment. Special emphasis is placed on visualizing track layouts, optimizing navigation strategies, and incorporating item avoidance mechanisms. The project leverages the `pystk2` library for enhanced track analysis and integrates advanced plotting techniques using Plotly to facilitate better visualization. The objective is to improve these agents and refine their performance before incorporating reinforcement learning techniques to enhance their capabilities.

1 Introduction

Our project aims to develop effective autonomous driving agents for *Super Tux Kart*. We began by implementing agents that apply various strategies for track navigation and obstacle avoidance. Initially, we developed the `StayInTheMiddleAgent`, an agent designed to maintain a central position on the track. However, the results were extremely poor, with both slow performance and unreliable behavior. Consequently, this agent will not be documented further, and we instead consider the `MedianAgent` as our first effective autonomous driving agent.

The following agents have been developed so far:

- `MedianAgent`: This agent uses an n-th node path-following approach to improve steering precision by targeting a future node rather than the immediate next one.
- `EulerAgent`: Focused on achieving maximum speed by refining acceleration and braking logic without relying on drifting mechanics.
- `ItemsAgent`: Introduced to test item avoidance logic, this agent incorporates peripheral vision to detect and avoid nearby items effectively.

To enhance our development process, we introduced visualization utilities that provide insights into track layouts, node positions, and kart trajectories. These utilities use the `pystk2` library alongside Plotly for detailed 3D plotting and enhanced debugging capabilities.

The project's next phase focuses on refining these agents to improve overall performance. Only after achieving sufficient reliability will we incorporate reinforcement learning techniques to further enhance decision-making in dynamic racing conditions.

2 Definitions

To provide clarity on key concepts used throughout this report, we define several important terms below:

- **Node:** A discrete point on the racing track that represents part of the path structure. Nodes are essential for determining the kart's intended direction and identifying key waypoints for navigation.
- **n-th Node:** In the context of our agents, the n-th node refers to a node that is a fixed number of steps ahead of the kart's current position. Targeting the n-th node (instead of the immediate next node) is a crucial technique to improve steering precision and minimize erratic movements.
- **Path Nodes:** The ordered list of nodes that outlines the entire track layout. This sequence of nodes provides the foundation for our agents' navigation logic.
- **Movement Vector:** A normalized vector that represents the forward-facing direction of the kart. It is calculated using the difference between the kart's position and its front-facing coordinate.
- **Dot Product:** A mathematical operation that evaluates the alignment between two vectors. In our project, the dot product between the movement vector and the direction to a given node is used to determine if that node lies ahead of the kart.
- **Height Difference (`height_diff`):** A condition used to filter nodes based on their elevation relative to the kart. This helps prevent selecting nodes that are incorrectly located on other track segments.
- **Peripheral Vision:** A method employed by the `ItemsAgent` that expands the agent's awareness to detect objects (like items) positioned slightly outside the kart's forward-facing direction.
- **Acceleration Logic:** The set of rules that control the kart's acceleration and braking. The `EulerAgent` emphasizes maximizing speed by carefully tuning this logic.
- **Obstacle Avoidance:** A mechanism employed by agents such as the `ItemsAgent` to identify and avoid items or track hazards. This is critical for improving race performance and ensuring the agent avoids penalties or collisions.

3 Analysis of Provided Scripts

This section provides a detailed overview of the scripts provided in the `src` folder. Each subsection discusses key algorithms, functions, and data structures to showcase our understanding of the codebase.

3.1 `__init__.py`

This script primarily handles environment registration and configuration. Key points include:

- **Environment Registration:** Several race environments are registered using the `gymnasium` library. Each environment specifies observation and action spaces to standardize agent interactions.

- **Wrapper Integration:** The script references multiple wrappers such as:
  - `ConstantSizedObservations`
  - `PolarObservations`
  - `FlattenMultiDiscreteActions`

  These wrappers modify observation data, adapt action space representations, and ensure stable inputs for learning models.

3.2 `definitions.py`

This file defines key data structures and observation-processing logic. The most important element in this script is:

- **`ActionObservationWrapper` Class:** This class modifies observations and actions before and after the environment's `step()` and `reset()` functions.

3.3 `envs.py`

The `envs.py` script is a fundamental component of the project that defines the primary racing environment, kart actions, observation spaces, and core game logic. This section presents a comprehensive understanding of its key classes, functions, and important algorithms.

3.3.1 `Phase` Enum

The `Phase` enumeration class defines the different stages of the race:

- `READY_PHASE`: Pre-race stage where the "Ready" sign is displayed.
- `SET_PHASE`: Pre-race stage where the "Set" signal appears.
- `GO_PHASE`: Start of the race when the "Go" signal appears.
- `RACE_PHASE`: The main race phase where karts are actively racing.

This class includes the method:

- **`from_stk()`:** Converts PySTK's native phase system into the custom `Phase` enumeration for improved compatibility with the project's environment logic.

3.3.2 Kart Action Space and Observations

Two critical functions define the environment's action and observation spaces:

- **`kart_action_space()`:** Defines the available kart actions such as acceleration, braking, drifting, firing items, and calling for rescue.
- **`kart_observation_space()`:** Constructs a comprehensive observation space containing kart data (position, velocity, orientation), item positions, track node distances, and other key metrics.

3.3.3 `STKAction` Class

This class defines a structured format for representing kart actions. Each action is represented as a dictionary containing:

- **Acceleration** — A continuous value between 0 and 1.
- **Steering** — A continuous value between -1 and 1.
- **Brake, Drift, Nitro, Rescue, Fire** — Boolean values controlling kart actions.

The associated function:

- **`get_action()`:** Translates the structured action data into PySTK's native format, ensuring compatibility with the racing environment.

3.3.4 `BaseSTKRaceEnv` Class

This core environment class extends the 'gym.Env' base class and manages key racing logic. Important attributes include:

- **`initialize()`:** Initializes the PySTK process and ensures the list of available tracks is loaded.
- **`reset_race()`:** Sets up a new race, randomizing track selection if none is specified.
- **`world_update()`:** Updates the race world state to keep observations accurate.
- **`get_state()`:** Extracts important game state data such as kart position, distance covered, and reward information.

3.3.5 Important Algorithm: Global to Kart Coordinate Conversion

The get_observation() method is a core function in the envs.py script, responsible for generating comprehensive observations for the agent. This method constructs the self.obs dictionary, which provides crucial environmental data for the agent's decision-making.

Purpose : The primary goal of get_observation() is to collect detailed environmental data about the kart, its surroundings, and track information — all in a structured format that the agent can efficiently interpret.

3.3.6 `STKRaceEnv` and `STKRaceMultiEnv` Classes

Both classes extend the core `BaseSTKRaceEnv` class and introduce additional functionality:

- **`STKRaceEnv`** — Designed for single-agent environments. The `reset()` and `step()` methods implement core episode logic such as resetting the kart's position, updating the race state, and calculating rewards.
- **`STKRaceMultiEnv`** — Designed for multi-agent races. This class manages multiple agents simultaneously, including their observation handling, action mapping, and synchronization during race steps.

3.3.7 Important Algorithm: Sorting Karts and Items by Distance

This algorithm sorts other karts and items in order of proximity, ensuring agents prioritize nearby obstacles for improved collision avoidance.

Algorithm 1 Sorting Karts and Items by Distance

Input: List of kart/item positions
Output: Sorted list of positions by proximity
$distances \leftarrow []$
for each $position$ in $positions$ **do**
    $distance \leftarrow \|position - kart\_position\|$
    Append $distance$ to $distances$
end for
Sort $positions$ in ascending order of $distances$
return $positions$

This method is critical for ensuring the agent responds promptly to immediate obstacles and racing threats.

3.3.8 Reward Calculation in `get_state()`

The reward function combines multiple factors:

- **Progress** — Based on the kart's forward movement along the track.
- **Position Bonus** — Reward for being higher in the race rankings.
- **Completion Bonus** — Reward for finishing the race early.

This multi-factor system incentivizes both speed and strategic positioning.

3.3.9 Summary

The `envs.py` script plays a vital role in managing SuperTuxKart's racing environment. It defines key environment logic, observation handling, and action management while implementing essential algorithms for navigation, item tracking, and race progression. These elements are fundamental to the success of the agents in achieving optimal racing performance.

3.4 `pystk_process.py`

This file handles SuperTuxKart's core process management. Key elements include:

- **`PySTKRemoteProcess` Class:** Manages communication between the Python environment and SuperTuxKart. The `run()` method serves as the race's core loop, where the environment is continuously updated, and agent actions are applied.
- **`PySTKProcess` Class:** Handles environment initialization, kart resets, and track loading. The `warmup_race()` method stabilizes the environment before racing starts.

3.5 `stk_wrappers.py`

This file introduces various observation and action wrappers designed to improve data flow between the environment and agents. Notable classes include:

- **`PolarObservations`:** Converts Cartesian coordinates to polar coordinates for simplified positional data.
- **Polar Coordinate Conversion Algorithm:** Used in the `PolarObservations` class to improve steering and directional logic.

Algorithm 2 Cartesian to Polar Conversion Algorithm

1: **Input:** Cartesian coordinates (x, y)
2: **Output:** Polar coordinates (r, θ)
3: $r \leftarrow \sqrt{x^2 + y^2}$
4: $\theta \leftarrow \arctan 2(y, x)$
5: **return** (r, θ)

3.6 `utils.py`

This file contains utility functions crucial for transforming environmental data. Key functions include:

- **`rotate()`:** Implements quaternion-based rotation, ensuring the kart's direction aligns with the environmental coordinate system.
- **`Discretizer` Class:** Facilitates discretization of continuous values to improve compatibility with learning models.
- **Global to Kart Coordinate Conversion Algorithm:** This algorithm is essential for transforming coordinates from the global reference frame into the kart's local view, improving obstacle detection and movement precision.

Algorithm 3 Global to Kart Coordinate Conversion

1: **Input:** Global coordinates (x_{global}, y_{global}), Kart position (x_{kart}, y_{kart}), Kart orientation angle θ
2: **Output:** Local coordinates (x_{local}, y_{local})
3: $x_{relative} \leftarrow x_{global} - x_{kart}$
4: $y_{relative} \leftarrow y_{global} - y_{kart}$
5: $x_{local} \leftarrow x_{relative} \cdot \cos(\theta) + y_{relative} \cdot \sin(\theta)$
6: $y_{local} \leftarrow -x_{relative} \cdot \sin(\theta) + y_{relative} \cdot \cos(\theta)$
7: **return** (x_{local}, y_{local})

This algorithm is particularly useful for aligning item positions, track nodes, and other dynamic elements in the kart's view for improved decision-making.

3.7 `wrappers.py`

This module provides wrappers designed to modify observation and action spaces. Key classes include:

- **`SpaceFlattener` Class:** Flattens complex observation spaces into simplified structures.
- **`MonoAgentWrapperAdapter` Class:** Adapts single-agent wrappers for multi-agent environments.

This comprehensive analysis reflects our understanding of the given codebase, highlighting key algorithms, data structures, and design principles crucial to the development of our autonomous agents.

3.8 Selecting the Node to target

To be able to take the n-th node in front of us, we must already know where the agent is on the track. The absolute position of the agent in the environment is given by

```
env.unwrapped.world.karts[0].location
```

and the list of all the nodes in the circuit is obtained with

```
env.unwrapped.track.path_nodes
```

For each node in this list, we follow the steps below:

Step 1.

```
agent_front = np.array(env.unwrapped.world.karts[0].front)
movement_vector = agent_front - agent_abs_pos
movement_vector /= np.linalg.norm(movement_vector)  #Normalisation
```

Determine where the agent is 'looking' (agent_front). Calculate the movement vector (movement_vector), which is the difference between the current position and the direction in front. Normalizes the movement vector to keep only the direction.

Step 2.

```
track_nodes = [np.array(segment[0]) for
               segment in env.unwrapped.track.path_nodes]
#path_start & path_end
```

List all the nodes of the track (track_nodes). Each segment[0] represents the starting point of a segment of the track.

Step 3.
Sorting nodes in front of the agent:

```
 nodes_ahead = []
for node in track_nodes:
    direction_to_node = node - agent_abs_pos
    node_distance = np.linalg.norm(direction_to_node)

    # Normalisation du vecteur direction
    if node_distance > 0:
        direction_to_node /= node_distance

    dot_product = np.dot(movement_vector, direction_to_node)

    # Vérifier la différence de hauteur
    height_diff = abs(node[1] - agent_abs_pos[1])

    # Vérifier si le nœud est devant l'agent et
    à une hauteur raisonnable
    if dot_product > 0 and height_diff < 5.0:
        nodes_ahead.append((i, node, node_distance))


direction_to_node = node - agent_abs_pos
# Calcule le vecteur de direction entre l'agent et le nœud.
dot_product = np.dot(movement_vector, direction_to_node)
```

Checks whether the angle between the agent's movement and the direction towards the node is positive. If $dot\_product > 0$, the node is in front of the agent. Filters out nodes that are too far apart in height ($height\_diff < 5.0$). Prevents the selection of nodes that are too high or too low in relation to the agent.

Step 4.
Sort remaining nodes by distance

```
    nodes_ahead.sort(key=lambda x: (x[0], x[2]))
    # Tri par distance croissante
```

Sorts the nodes_ahead list according to the distance between the agent and each node. The aim is to select the second closest node.

Step 5.
Selecting the second closest node

```
    if len(nodes_ahead) > 1:
    # Sélection du deuxième nœud le plus proche
    second_node_pos = nodes_ahead[1][1]
elif len(nodes_ahead) == 1:
    # Si un seul nœud disponible, on le prend
    second_node_pos = nodes_ahead[0][0]
else:
```

```
# Si aucun nœud valide, on garde la position de l'agent
second_node_pos = agent_abs_pos
```

Step 6.
The vector l between the agent and the second node is calculated

```
vector = second_node_pos - agent_abs_pos
```

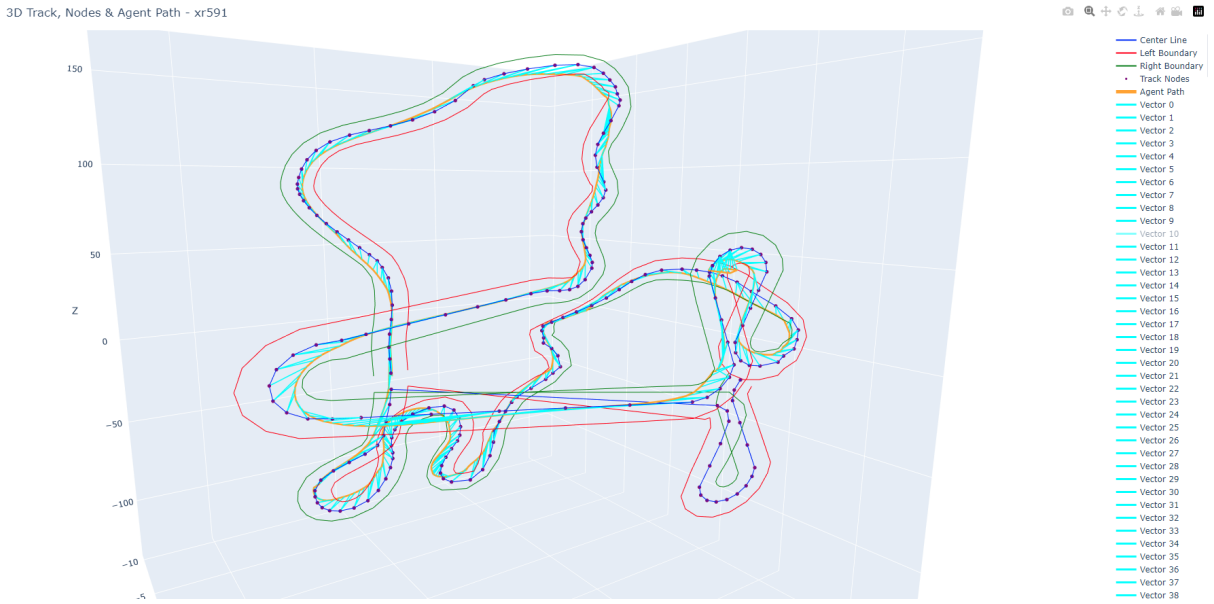

Figure 1: Track xr591 with 2 nodes in front of the agent

but if we choose 3 nodes in front of the agent, the algorithm can choose nodes that are not on the same segment of the track as the agent:

This is because we only consider the distance and the fact that the node is in front of it, but do not take into account whether the node is on another segment of track. That's why we introduced the height_diff test, but it needs to be corrected because it doesn't really work at the moment for cases with more than 2 nodes in front of it. It's a start, but it needs to be corrected...

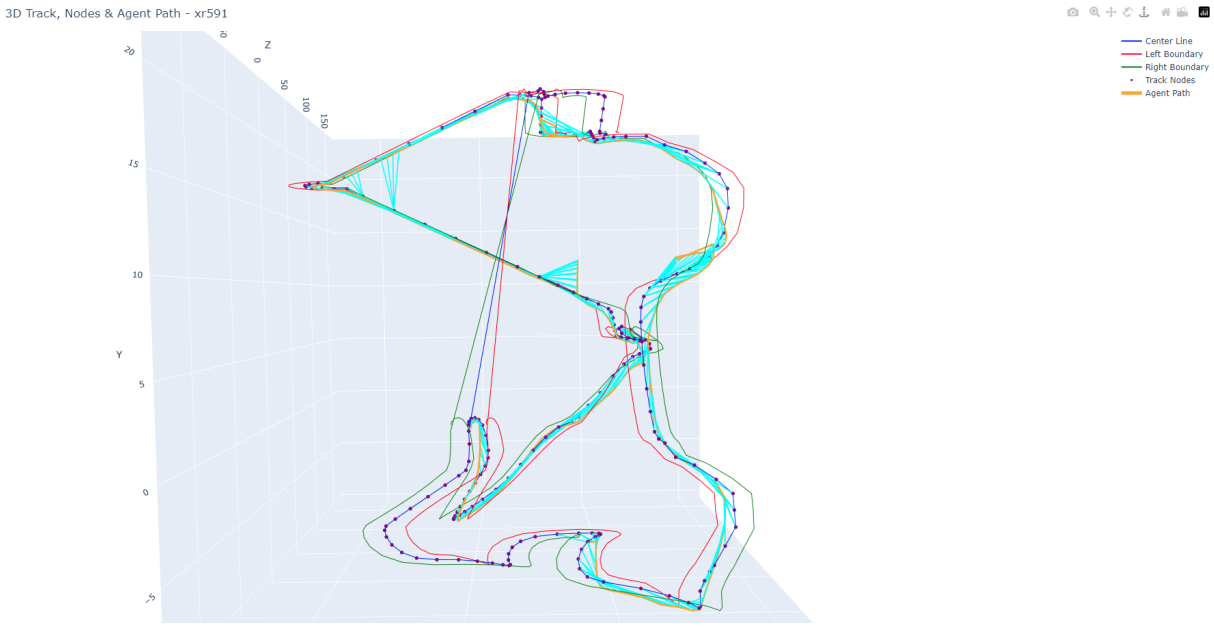

Figure 2: Track xr591 with 3 nodes in front of the agent

Algorithm 4 ComputeSecondNodeVector Algorithm

Input: Agent's position $agent\_abs\_pos$, Kart's front position $agent\_front$, List of track nodes $track\_nodes$
Output: Vector from the agent to the second closest node ahead
Compute $movement\_vector \leftarrow agent\_front - agent\_abs\_pos$
Normalize $movement\_vector \leftarrow \frac{movement\_vector}{\|movement\_vector\|}$
Initialize $nodes\_ahead \leftarrow []$
for each $node$ in $track\_nodes$ **do**
    Compute $direction\_to\_node \leftarrow node - agent\_abs\_pos$
    Compute $node\_distance \leftarrow \|direction\_to\_node\|$
    if $node\_distance > 0$ **then**
        Normalize $direction\_to\_node \leftarrow \frac{direction\_to\_node}{node\_distance}$
    end if
    Compute $dot\_product \leftarrow dot(movement\_vector, direction\_to\_node)$
    Compute $height\_diff \leftarrow |node.y - agent\_abs\_pos.y|$
    if $dot\_product > 0$ **and** $height\_diff < 5.0$ **then**
        Append $(i, node, node\_distance)$ to $nodes\_ahead$
    end if
end for
Sort $nodes\_ahead$ by $(x[0], x[2])$ in ascending order
if length of $nodes\_ahead > 1$ **then**
    $second\_node\_pos \leftarrow nodes\_ahead[1][1]$
else if length of $nodes\_ahead == 1$ **then**
    $second\_node\_pos \leftarrow nodes\_ahead[0][0]$
else
    $second\_node\_pos \leftarrow agent\_abs\_pos$
end if
Compute $vector \leftarrow second\_node\_pos - agent\_abs\_pos$
return $vector$