1. Introduction

For this project, you will implement a simple 2D racing video game, by precisely following the instructions given in this document.

The goals of the project are:

- Practice the OOP concepts studied in class
- Gain some experience in python
- Gain some experience in designing algorithms
- Learn to collaborate on programming projects

The project will be evaluated based on:

- The deliverables:
  - An UML class diagram modeling the project
  - The python implementation corresponding to the diagram
- An oral examination about the project
- A competition between your AIs

Schedule:

- 30/10: Formation des groups of three (random affectation for remaining students)
- 17/12: Delivery of the class diagram and python implementation
- Semaine du 18/12: Oral examination

The program you have to write is a 2D video game based on the pygame library. This library makes it possible to easily build graphical interfaces in which you can draw 2D objects. A starting point for the implementation is provided to you. It contains classes definitions that you will need to round off. You are obviously encouraged to add more classes, attributes and methods, in order to make your implementation fit the UML class diagram you will design.

Be careful however, there are some parts of the starting pack that you should not edit:

- In the file `track.py`, the definition of the class `Track` should not be modified. You have to make it so the rest of the code you write is compatible with the definition of this class. You will be penalised if your code is not compatible.
- In the file `main.py`, the last lines' role is to launch the game. You should not modify those lines.
- Some variables are defined in upper case. You should not change the value of those variables, they allow to standardise your implementations. If you change those variables, your game's physics will be different from mine. If your AI works well on your game, but your game is different from mine, then your AI might not work properly on mine. To avoid any issue, do not modify the values of the upper case variables.
- The file `test.py` and the folder `test/`. See section 6.

2. Graphical User Interface (GUI)

The GUI implementation is based on pygame. You can import this library at the start of your files, and use it to draw objects.

The track you will have to draw is represented by a string (example in `main.py`). This string corresponds to a 2D grid (with line breaks), with each character associated to a type of surface.

Each type of surface has an associated class, for which you will have to define a method `draw`, that should draw the surface on the screen. This method takes as input a `screen` object and should draw a square shape on the screen at the right position.

The track is thus described as a grid which blocks (of size `BLOCK_SIZE`) are defined by the characters (B, C, D, E, F, G, L, and R). The character B corresponds to a block `Boost`, the characters C, D, E, F correspond to `Checkpoint` blocks, with different identifiers. The character G corresponds to a `Grass` block. The character L corresponds to a `Lava` block. The character R corresponds to a `Road` block. The following figure provides an example display of a track and the corresponding string.

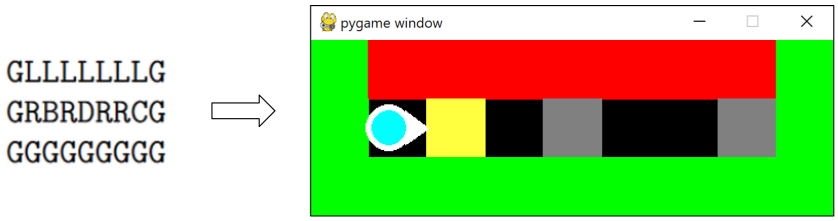

Figure 1. Example of string used to describe a track, and corresponding GUI.

This is an example of display that you can obtain after having properly implemented the `draw` methods of all the different surfaces. You can aim for a visual completely different from this one, you don't event have to have plain colours, you could for instance try to add textures or images. With the starting pack, only a circle is drawn on the screen, this is because most of the implementation of the GUI is missing and need to be written by you.
To implement the GUI, a good starting point is to take a look at the pygame documentation : https://www.pygame.org/docs/ref/draw.html
Finally, note that in this example, the first checkpoint is the grey square on the right (corresponding to the letter C) and the second and final checkpoint is the grey square in the centre (corresponding to the letter D).

3. Game physics

The game physics correspond to a set of rules governing the motion of the kart, and its interaction with the different types of surface. First, the kart is controlled by four commands (Forward, Backward, Left, Right) that can be simultaneously activated.
At each iteration of the `while` loop in the `Track` class, a command is received by the kart (from the human player or the AI). This command modified the acceleration and orientation of the kart according to the following rules:

- Forward: $a_c(t) = a_{max}$
- Backward: $a_c(t) = -a_{max}$
- Left: $\theta(t) = \theta(t-1) - v_\theta$
- Right: $\theta(t) = \theta(t-1) + v_\theta$

If the Forward and Backwards commands are activated at the same time, or if none of them is activated, then $a_c(t) = 0$. We parameter by t the different quantities that vary over time: t corresponds to the new value, and $t-1$ corresponds to the past value. a_{max} and v_θ are parameters for which you should absolutely respect the given values:

- $a_{max} = 0.25$
- $v_\theta = 0.05$

These values are reminded in the starting pack files.

The default surface type is `Road`. On this surface type, the kart motion follows the following rules:

$$\theta_v(t-1) = \arctan\left(\frac{v_y(t-1)}{v_x(t-1)}\right) \tag{1}$$

You can use the `atan2` function from the `math` module.

$$|v(t-1)| = \sqrt{v_x(t-1)^2 + v_y(t-1)^2} \tag{2}$$

$$a(t) = a_c(t) - f * |v(t-1)| * \cos\big(\theta(t) - \theta_v(t-1)\big) \tag{3}$$

$$v(t) = a(t) + v(t-1) \tag{4}$$

$$v_x(t) = v(t)\cos(\theta(t)) \tag{5}$$

$$v_y(t) = v(t)\sin(\theta(t)) \tag{6}$$

$$x(t) = x(t-1) + v_x(t) \tag{7}$$

$$y(t) = y(t-1) + v_y(t) \tag{8}$$

These rules define the next position of the kart according to its past position, past velocity, past acceleration, and current orientation. We use a friction coefficient $f = 0.02$.

The surface types `Grass`, `Boost` and `Lava` have an influence on the motion of the kart:

- `Grass`: On this surface type, everything goes the same as on `Road`, except for the value of the friction coefficient that becomes $f = 0.2$ in equation 3.
- `Boost`: On this surface type, the velocity $v(t)$ is no longer derived from equations 3 and 4, but is instead given by $v(t) = v_{boost} = 25$.
- `Lava`: The kart falls and reappears at the coordinates of the last checkpoint. In practice, we just modify the position, orientation and velocity of the kart according to the rules:

$$\begin{aligned} x(t) &= x_c \\ y(t) &= y_c \\ \theta(t) &= \theta_c \\ v_x(t) &= 0 \\ v_y(t) &= 0 \end{aligned}$$

  where x_c, y_c and θ_c correspond to the position and orientation of the kart when it passed the previous checkpoint.

Finally, the surface type `Checkpoint` saves the progression of the kart on the track. The `Checkpoint` objects have an attribute id ranging from 0 to 3 that can be used to identify them. When the kart is on a surface of type `Checkpoint`, we have three options:

(1) If the checkpoint id does not match the next checkpoint the kart has to reach, then nothing happens (this is not the right checkpoint).
(2) If this is the correct id, and moreover this is the last checkpoint of the track, then the race is over, and we set the attribute `has_finished` of the kart to `True`.
(3) If this is the correct id, but not the last checkpoint, then we record the current position and orientation of the kart, and update the id of next checkpoint for the kart:

$$\begin{aligned} x_c &= x(t) \\ y_c &= y(t) \\ \theta_c &= \theta(t) \\ \text{next\_checkpoint} &+= 1 \end{aligned}$$

Important remarks:

(1) The outside of the track must behave like lave.
(2) We consider that the effects of a surface type are applied to the kart only if the kart position (x, y) is situated on a pixel of the corresponding surface. Thus, it is possible that a part of the kart appears to be touching a type of surface without the kart being affected by the corresponding effects.

4. Starting pack

A starting point of the implementation is provided. The code contains:

- Files `boost.py`, `checkpoint.py`, `grass.py`, `lava.py` and `road.py` in which you will have to implement the different surface types.
- A file `kart.py` in which you will have to implement the behaviour of the kart.
- A file `track.py` implementing the unfolding of the game. **You should not change this file.**
- A file `human.py` implementing a human player. This file allows you to directly control the kart with the keyboard, which should help you testing your implementation.
- A file `ai.py` implementing an AI. A first version is given to you as a starting point. This AI is full of flaws: for instance, it does not even try to avoid lava or grass. It's up to you to improve it!
- A file `main.py` instantiating a `Track` object from a string description of the track, and launching a game. You can modify the beginning of this file as you please to experiment with different tracks.
- A file `test.py` as well as a `test/` folder with several files. **You should not change those.**

Be careful, it is highly probable that other classes appear in your UML class diagram. You should obviously implement those. Do not limit yourselves to the files already present in the starting pack.

5. UML diagram

(1) Your UML diagram should model the project while using all the object-oriented principles (links between classes, encapsulation, abstraction, polymorphism).
(2) The project allows the use of these concepts and your diagram should therefore contain:
    - at least one aggregation or composition
    - at least one inheritance link
    - at least one abstract class
(3) The choices of classes links (other than inheritance) and encapsulation must be justified (in a few lines).
(4) The UML class diagram must be readable, and it is therefore recommended to draw it:
    - with the help of a software (Dia, ArgoUML, . . .)
    - via a website (online.visual-paradigm.com, creately.com/lp/uml-diagram-tool)

You should deliver a pdf file containing your UML class diagram as well as the required justifications.

6. AI Competition

The AI you will implement will be put in competition with the others, on tracks that are not yet disclosed. Your implementation should still follow certain rules:

(1) Avoid using external python libraries. The only authorised libraries are those already imported in the starting pack, and `numpy`. If you want to use other libraries, you should first ask me by email.
(2) Your implementation should not be too slow. You will be penalised if the computations take too long (aim for less than 100ms per call of the `move` method).

7. Tests

The file `test.py` contains a script that will help you checking whether you are on the right path. The script automatically generates games with different test tracks, and pre-recorded commands. If your implementation is correct, the script should simulate a full game, and display:

```
==================== VOTRE CODE A PASSE LE TEST ====================
```

If the test fails, it should display:

```
========================== ECHEC DU TEST ===========================
```

Five test scenarios are available:

- `un_checkpoint`: For your code to pass this test, you have to have correctly implemented the kart physics on the `Road` surface type, as well what happens when the kart reaches a `Checkpoint`.
- `plusieurs_checkpoints`: This scenario presents a more complex situation with four checkpoints. To pass this test, your code has to properly manage the order of the checkpoints.
- `grass`: To pass this test, your code has to properly implement the behaviour of the kart on `Grass`.
- `boost`: To pass this test, your code has to properly implement the behaviour of the kart on a `Boost`.
- `lava`: To pass this test, your code has to properly implement the behaviour of the kart when it reaches `Lava`, as well as its behaviour when it goes outside of the track.

To run a test, you should run the script from your command line/terminal. For instance, for the first scenario:

```
python test.py un_checkpoint
```

You can replace `un_checkpoint` with other test scenarios.

You don't necessarily have to use the tests, they are just here to help you checking that your implementation is compatible with what is expected.