Sonic Platformer Physics Challenge

Challenge Preamble

In this challenge, you will implement the physics and control system of a sideway 2D platformer, similar to the classic Sonic the Hedgehog game. The goal is to simulate Sonic's movement across a map composed of solid tiles and empty spaces. The player's movement is affected by gravity, friction, and velocity limits. Each frame applies one control command (`LEFT, RIGHT, JUMP, or WAIT`), and you will compute Sonic's position according to the laws of motion.

Part 1 – Platformer Physics: Theory

Before diving into code, let's explore the physics model that will drive Sonic's movement. In a platformer, movement occurs on a discrete time step — typically one update per frame. Each frame, we apply forces such as player input, gravity, and friction. We then calculate the new position using the current velocity and handle collisions with the map.

Each tile on the map represents either solid ground (`'#'`) or empty air (`'.'`). The player is represented as a single point moving within this grid. The map also contains a single starting point marked as `'S'`. Out-of-bounds areas are considered solid walls to prevent the player from moving off-screen.

Physics Constants

Constant	Value / Description
`GRAVITY`	0.06 → Gravity constant (pulls Sonic down each frame, if he is in the air).
`FRICTION`	0.80 → Reduces horizontal velocity each frame by this factor.
`VX_MAX`	0.5 → Maximum horizontal velocity magnitude. (on both side : plus and minus)
`VY_UP_MAX`	-0.6 → Maximum upward (jump) velocity magnitude.
`VY_DOWN_MAX`	0.8 → Maximum downward (falling) velocity magnitude

Part 2 – Object-Oriented Design

This challenge is the first one in our series to introduce Object-Oriented Programming (OOP). OOP is a way of structuring programs around objects — entities that combine **data (attributes)** and **behavior (methods)**. This approach allows our game to be modular and easier to extend. For example, Sonic is an object with properties such as position, velocity, and state (on ground or in air), and methods such as `manage(self, command)` to update physics and '`print_debug(self)`' to display the current state.

You'll notice that each methods of a class (contrary to python function) all have a "`self`" as a mandatory first argument. This is to differentiate between different "**instances**" of the object.

The two main classes implemented here are `SonicMap` and `SonicPlayer`. `SonicMap` handles the environment: loading the level, finding the spawn point, and testing for walls. `SonicPlayer` handles the physics: reading commands, applying gravity, friction, and movement. Both classes work together to simulate the game world.

Part 3 – Code Walkthrough

File: `SonicMap.py`

The **`SonicMap`** class is responsible for loading the ASCII map and managing collision detection. It reads the file line by line, strips newlines, and stores the result as a list of strings. The method **`find_start(self)`** searches for the character 'S' to locate the player's initial coordinates. Finally, **`is_wall(self, row: int, col: int)`** checks if a tile is solid (**`'#'`**) or out of bounds.

File: SonicPlayer.py

This file defines the **`SonicPlayer`** class, where most of the physics are implemented. The constructor (**`__init__(self, mapFile)`**) initializes all attributes: position, velocity, and state. Constants such as gravity (G), friction, and velocity limits are defined at the top of the file. The **`manage(self, command)`** method executes the core logic each frame:

1. Apply command impulse (**`LEFT, RIGHT, JUMP, WAIT`**), using the value as the force. That means that:
   - **`LEFT 0.10`** will subtract (moving left) 0.10 to **`x_velocity`** variable.
   - **`RIGHT 0.10`** will add to the **`x_velocity`** variable
   - **`JUMP 0.50`** will subtract 0.5 from the **`y_velocity`** variable. Ensuring it doesn't lower than **`VY_UP_MAX`**
   - **`WAIT 1`** will do nothing for this loop (you can ignore the value 1)

2. Apply gravity to vertical velocity. By adding the constant to the **`y_velocity`** variable, ensuring it doesn't go over **`VY_DOWN_MAX`**.

3. Apply friction to horizontal velocity, by multiplying the **`x_velocity`** variable by the **`FRICTION`** factor.

4. If not done already, clamp both velocities to avoid unrealistic speeds.

5. Move horizontally, then vertically, using collision detection.

The methods **`_move_horizontal(self)`** and **`_move_vertical(self)`** handle movement along each axis separately. If a collision is detected, Sonic's position is corrected to the nearest free space and the corresponding velocity is reset. When Sonic lands on a surface, the '**`on_ground`**' flag becomes True, allowing jumps on the next frame. This boolean is use to determine whenever Sonic can jump. he cannot activate a jump while already in mid-air (No Castlevania-style double-jump here!).

File: e2b.py

This is the main entry point of the challenge. It initializes the **`SonicMap`** and **`SonicPlayer`** objects, loads the command list from a file, and iterates through each command, updating Sonic's state. The loop calls **`player.manage(self.command)`** and **`player.print_debug(self)`** to display the frame's position, velocity, and status. This script effectively drives the simulation using pre-recorded inputs.

Part 4 – Stretch Goal: Visualization

For students who want to go further, a stretch goal can be to integrate the Pygame library for visualization. By implementing a draw() method in SonicPlayer, you can render Sonic and the map in real-time. This visual feedback helps understand how gravity and collisions affect motion, and makes debugging more intuitive.

Part 5 – Summary

This challenge introduced several key programming and physics concepts:

- Object-Oriented Programming (classes, attributes, methods)
- Physics simulation with gravity and friction
- Tile-based collision detection
- File-driven command input for deterministic testing

By the end of this lab, you should understand how to structure a small physics-based game using OOP principles and be able to extend it with additional mechanics such as acceleration, slopes, or multiple players.

If you succeed creating this simulation, **Sonic should start the simulation on tile (58, 18) and end on tile (58, 2)**