Shooter – Moving Targets Challenge

Challenge Preamble

This challenge pushes your understanding of Object-Oriented Programming (OOP) in Python. You will create a dynamic shooting simulation where multiple moving targets approach or bounce within a bounded field, while a player fires instantaneous laser-like shots ('raygun') at them. The entire simulation is displayed using the **`Pygame`** library, offering a real-time view of the targets' movement, shots, and score evolution.

The key learning objective is to structure a **`Pygame`** project into multiple files and classes, each responsible for a specific role: configuration, game loop, entity behavior, and rendering. The code organization you will practice here reflects how actual video game projects are structured for scalability and maintainability.

Part 1 – Concepts and Behavior

1.1 Target and Bullet Dynamics

Each frame, the Game updates all Target objects. The update method applies the velocity scaled by delta time. If a target reaches the limits of the world defined in **`Config`** class, it reverses its direction, effectively bouncing off the boundary.

When a bullet is fired, the Game immediately checks which targets are within the hit radius. The distance calculation uses the Euclidean formula **`sqrt((dx)^2 + (dy)^2)`** or **`math.hypo()`**. Each target within this radius is considered hit, marked inactive, and contributes to the score.

1.2 Main Loop and Timing Logic

The game loop runs at 60 FPS, controlled by **`pygame.time.Clock()`**. Every iteration updates motion based on the frame's delta time. A step counter triggers every second (1000 milliseconds), pulling the next instruction from InstructionStream. This ensures one shot per second, regardless of framerate.

1.3 Rendering System

Rendering occurs after each update cycle. The field background, concentric rings, and all active targets and bullets are drawn in world-to-screen coordinates. The HUD displays current time, number of steps, shots fired, hits, and total score.

1.4 Integration and Flow

At startup, **`main.py`** loads data files into memory, instantiates all class objects, and begins the main loop (the **`run()`** method of the **`Game`** object). The simulation proceeds for as long as the user keeps the window open or until all instructions are processed. At the end of execution, the final score and shot statistics are printed to the console.

Part 2 – Project Architecture and Code Walkthrough

This project consists of several Python modules, each with a clear and single responsibility. Together, they form a coherent architecture for the shooting simulation.

1.1 `Config.py` – Centralized Configuration

The **`Config`** class acts as the central configuration hub for the entire program. It defines constants for screen size, world coordinates, colors, physics constants, and scoring. Having a dedicated configuration class ensures that all parts of the game share consistent parameters.

Key attributes to include:

- **`WORLD_MIN/WORLD_MAX`**: define the playable world boundaries in world units. Set these to **-1000** and **1000**
- **`SCREEN_W/SCREEN_H`**: **`Pygame`** display resolution. You can set your own values for this it while stretch the displaying windows. Mine was set at **900, 900.**
- **`STEP_DELAY_MS`**: defines the timing for shots (one per second). Set to **1000**.
- **`TARGET_RADIUS:`** defines the radius of the Target in world coordinate. Set to **40.**
- **`BULLET_RADIUS_PX`**: determine the size of the visual entities. This is in Screen coordinate. The real world bullet is only 1 world unit, but for easy of visualization we render it a bit bigger so that we can see it. Set to **5.**
  `SCORE_PER_HIT`: define game logic constants for the score of each target. Set to **10**.

1.2 `UtilityFunctions.py` – Coordinate Conversion

This helper module converts between 'world space' (logical coordinates used by the simulation) and 'screen space' (pixels on the window). It includes functions for translating positions and radius sizes (**`world_to_screen()`** and **`world_radius_to_pixels()`** methods), ensuring all elements are drawn proportionally to their world values.

1.3 `Target.py` – The Moving Entities

Each target is represented by an instance of the Target class. Targets have an initial position (x, y), a velocity vector (vx, vy), and a color. During the simulation, each target updates its position continuously based on velocity and delta time.

When a target hits a boundary of the world, it bounces back by inverting its velocity. Targets can be hit by bullets: when that happens, their color changes and they are marked as inactive.

Beside the **`__init__()`** constructor method, the class should have **`draw()`** method to draw the target at the screen using the **`pygame.draw.circle()`** function and **`update()`** methods to adjust the position of the object based on its velocity. This method also manage the bouncing off the world boundaries. It could also be a wise idea to include a **`distance_to()`** method receiving 2 arguments (x, y) to test collision between this target and a point (bullet).

1.4 `Bullet.py` – Instantaneous Shots

A Bullet is a simple class representing a 'raygun' shot. Unlike typical projectiles, bullets do not move. They appear instantly at the targeted coordinate. Each shot is drawn as a red circle, and collision detection is performed immediately after firing.

Since this object doesn't move, there is no need for an update() method. So just a constructor (**`__init__()`**) and **`draw()`** method would be needed.

1.5 `InstructionStream.py` – Reading the Firing Sequence

This class provides a sequential stream of instructions, feeding one line per second to the **`Game`** class. It abstracts how shooting instructions are read and managed, allowing for future extensions like random generation or real-time input.

Just a constructor and a **`next()`** method are sufficient here.

1.6 `Game.py` – The Core Engine

The **`Game`** class is the heart of the program. It initializes **`Pygame`**, loads targets and instructions, and runs the main game loop.

The **`update()`** method continuously moves targets, processes shots once per second, and checks for collisions between bullets and targets.

The **`draw()`** method handles the display of the moving targets, the field boundaries, the HUD (time, score, hits), and the bullet positions. By separating the game logic and rendering code, the design remains clean and modular.

Beside that there should be a **`run()`** method that will be the main loop of the game. The rest is up to you based on how you want your code to be divided.

1.7 `main.py` – The Entry Point

This script loads the initial data files (targets and shots) and creates the **`Game`** object. It serves as the entry point for running the entire simulation. This will be the easiest file ever, basically in the **`main()`** function, start a **Game** instance and **`run()`** it. Whether you do the loading of the file here or in the Game class is up to you. I actually did it here, and pass the **`targets`** and **`instructions`** as lists in memory to the **`Game`** instance

Conclusion

This challenge showcases how to organize an Object-Oriented **`Pygame`** project and manage real-time simulation. You have implemented continuous motion, collision detection, modular architecture, and time-based logic. Completing this exercise demonstrates readiness to tackle advanced **`Pygame`** mechanics such as sprite animation, projectile trajectories, and event-driven input systems.

If you succeed you challenge and wait for 100 seconds to watch the game-play, you should end up with a score of **120pts.**