Game in Python

Jakub Dybich, Krzysztof Jabłoński, Mateusz Wardawa, Sebastian Zarębski, Jan Włodek

March 9, 2025

1 Introduction

1.1 Purpose of the project

The goal of our project was to develop a simple platform shooter game in Python and implement artificial intelligence (AI) bots capable to play the game using reinforcement learning (RL). The project was divided into two main phases:

- **Game Development** - Create fully functional two-dimensional platform shooter game with mechanics such as shooting, player movement, physics, and bot entities.
- **Reinforcement learning for bots** - Train AI controlled agents to play the game using reinforcement learning techniques.

1.2 Used Technologies

We used following technologies in our project:

- Python - the programming language used in the project.
- PyGame - a library used for handling graphics and user input.
- PyTorch - a library used to implement reinforcement learning.
- OpenAI Gym - used for handling reinforcement learning environment.

1.3 Project structure

The project is organized into multiple directories, each serving a specific purpose. Below is shown an overview of the structure and descriptions of key files.

```
python game (Root directory of the project)
├─characters (Contains implementations of game entities)
│ ├─bots.py (Defines bot class and its behavior)
│ ├─bullets.py (Manages bullet mechanics)
│ ├─entity.py (Base class for all game entities)
│ ├─player.py (Handles player movement, shooting, and interactions)
│ └─weapon.py (Defines different weapon classes, shooting and
│    reloading mechanics)
├─game (Core game mechanics and rendering)
│ ├─renderer.py (Handles game rendering and graphics)
│ └─simulation.py (Manages game loop and all simulations)
├─map (Contains map and obstacle classes)
│ ├─map.py (Defines the game map structure and platform placement)
│ └─obstacle.py (Defines obstacle classes)
├─machine_learning (Contains RL implementation for bot training)
│ ├─dql_agent.py (Defines the Deep Q-Learning agent)
│ ├─environment_test.py (Tests the RL environment setup)
│ ├─environment.py (Creates the training environment for bots)
│ └─register_env.py (Registers custom environments for training)
├─ml_saves (Stores trained models and RL checkpoints)
├─config.py (Stores game settings and configuration parameters)
├─main.py (Main entry point to run the game)
└─train_agent.py (Script to train the RL agent)
```

Figure 1.1: Project Directory Structure

2 Game developement

2.1 Game Concept and Mechanics

The project involves developing a 2D platform shooter game where players navigate through the map, eliminating enemies while avoiding obstacles. Key gameplay elements include:

- Player movement - running and jumping.
- Shooting mechanics - various weapon types and ammunition management.
- Enemies
- Collision detection - ensure that bullets, enemies, players and obstacles interact properly.

2.2 Game architecture

2.2.1 Renderer and Simulation

Renderer module is responsible for initializing the Pygame window, listening for user input, forwarding input events to the Simulation object, and rendering game objects such as the player, enemies, obstacles, etc.

On the other hand, the Simulation module handles the game logic by processing it, updating the positions of entities, checking for collision events, and updating the overall game state.

The application can operate in two modes:

1. **Graphical mode** - the mode with initialized game window that listens for user input and renders updated game state every frame
2. **Simulation mode** - the mode that runs simulation without initializing Renderer. This mode will be used in machine learning algorithms.

2.2.2 Entity class

Entity is an abstract base class that defines the fundamental attributes and behavior shared by all game objects. It represents objects with a defined position (x and y positions of the object's center) and size in the game world (height and width) and provides common methods such as draw, check_collision or evaluate_collision. The following classes inherit from entity:

- Player - user-controlled character able to move, jump and shoot.
- Bot - character able to move and shoot.
- Obstacle - stationary object representing element of a map.
- Bullet - a moving projectile fired by the player or bots, interacting with objects upon impact.

2.2.3 Map class

The Map class, implemented in map.py, defines the layout of the game environment, map width, map height and list of platforms. It loads predefined map.

2.2.4 Weapon class

The Weapon class, from module weapon.py, is an abstract base class that defines the behavior of firearms, including shooting mechanics, reloading, and ammunition management.

2.2.5 Configuration

The config.py module centralizes all game settings, allowing quick modifications without altering the main code. It contains various configuration classes, such as PlayerConfig, BulletConfig, and BotConfig.

We decided to use dataclasses with the parameter `frozen=True`, which prevents accidental modifications of constants. Dataclasses proved to be very useful, as they allowed us to use a clean and intuitive syntax like `ClassName.property`.

2.3 Implementation details

2.3.1 Player and movement system

The player character has the following movement capabilities:

- Can move left and right using the **A** and **D** keys.
- Can jump using the **spacebar** or **W** key.
- Falls with a fixed acceleration directed downward if there is no surface directly below.
- Can pass through platforms from below when jumping, allowing the player to move upward without obstruction.

The physics system calculates movement speed with the following equation:

$$new\_velocity = velocity + acceleration \tag{2.1}$$

Gravity is applied every frame, affecting vertical acceleration.

2.3.2 Bots' Behavior

We have implemented three types of bots: **Bot (default)**, **Sprinter**, and **PlayerLikeBot**. The difference between the first two is that the Sprinter bot continuously accelerates toward the player, starting from zero velocity and reaching the speed defined in the configuration within a certain period. When the Sprinter reaches its target, it loses all its speed and has to accelerate again.

The default bot, on the other hand, moves randomly but tends to head toward the player. Its mean movement angle is set to 0° (directly toward the player) with a standard deviation of 80°. It moves slightly slower than the player, whereas the Sprinter's maximum speed is slightly higher.

As the name suggests, the **PlayerLikeBot** behaves like a player but is not controllable. Its movement and jumping frequency are entirely random. Our goal was to program these bots to learn how to play the game so that we could eventually play against them.

All bots were able to shoot, but only PlayerLikeBot did not attempt to shoot the player — this was also part of the planned training process.

2.3.3 Shooting and weapon system

- The player can shoot using the **left mouse button**. The direction of shoot is indicated by mouse position.
- There are three types of weapons each of them has different properties, including fire rate, spread angle and magazine capacity.
- Bullets travel in a straight line from the player's position in the direction they were fired.
- Bullets can hit enemies, dealing damage upon impact.
- Bullets collide with obstacles and map borders they are destroyed after collision with them.
- The player can switch between available weapons using keys **1**, **2** and **3**.
- Weapons have ammunition, pressing **R** key cause reloading weapon.

2.3.4 Collision detection

All objects in the game are rectangular in shape, and the following function is used to indicate collisions between them

```python
def check_collision(self, other):
    return (
        abs(self.x - other.x) <= self.width / 2 + other.width / 2
        and abs(self.y - other.y) <= self.height / 2 + other.height / 2
    )
```

where x and y are coordinates of object's center.
The outcome of this check may trigger different actions depending on the classes of the objects involved.

2.4 Graphical user interface

The following image presents a screenshot, illustrating the graphical user interface and the elements of a game:

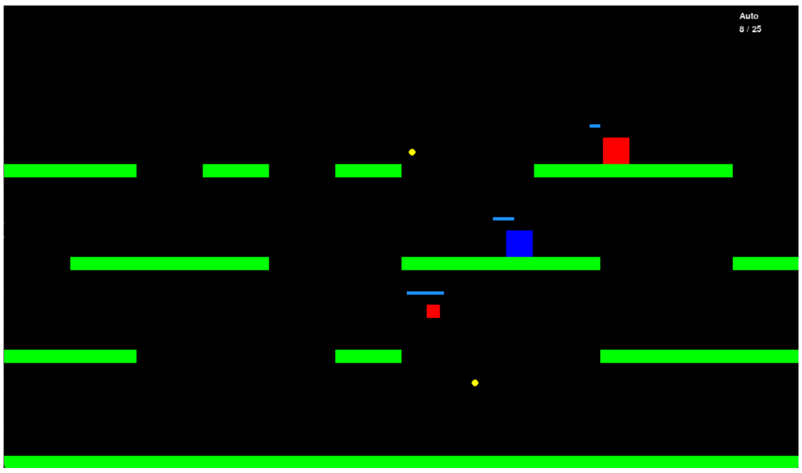

Figure 2.1: Graphical user interface for the game

The interface consists of several key elements:

- Player character (blue rectangle) - the user-controlled entity that can jump, move and shoot.
- enemies (red rectangles) - entities able to move and shoot
- bullet (yellow rectangle) - Projectiles fired by player or enemy.

- health bars (thin light blue rectangles above enemies and player) - health point indicators.
- weapon info (top right corner) - information about weapon type, ammunition and magazine capacity.

2.5 Testing

A crucial part of game development was to ensure that all the mechanics worked correctly. Since the game involves real-time interactions, manual testing was the most effective way to verify gameplay mechanics.

3 Machine learning

3.1 Environment

In this part, we created an environment class compatible with the OpenAI Gymnasium module.

3.1.1 Action space

```
self.action_space = Dict(
    {
        "move": Discrete(3),  # 0: no move, 1: right, 2: left
        "jump": Discrete(2),  # 0: no jump, 1: jump
        "shoot": Discrete(2),  # 0: no shoot, 1: shoot
    }
)
```

This implementation lacks the ability to choose the shooting angle. A potential solution is to implement an additional action called "shoot_angle" as follows:

```
"shoot_angle": Box(
    low=0, high=2 * np.pi, shape=(), dtype=float
)
```

However, this creates a hybrid action space, which is not well-suited for algorithms like DQN. Another option is to define our action space as MultiDiscrete:

```
self.action_space = MultiDiscrete([[3, 2, 2, 36]])
# Shooting angle is discretized into intervals of 10 degrees
```

While this is still not ideal for DQN, it works well for algorithms such as A2C or PPO.To make the action space compatible with DQN, we should define it as Discrete:

```
self.action_space = Discrete(41)
# Actions are:
# 1: Go right
# 2: Go left
# 3: Jump
# 4-41: Shoot at angles (0° to 360°, in 10° intervals)
```

3.1.2 Observation space

```python
self.observation_space = Dict(
    {
        "actor": Dict(
            {
                "position": Box(
                    low=-np.inf, high=np.inf, shape=(2,), dtype=np.float32
                ),  # (x, y)
                "health": Box(low=0, high=100, shape=(), dtype=np.float32),
                "ammo": Box(low=0, high=100, shape=(), dtype=np.float32),
                "isReloading": Discrete(2),  # 0 or 1
            }
        ),
        "obstacles": Box(
            low=-np.inf,
            high=np.inf,
            shape=(self.max_obstacles, 4),
            dtype=np.float32,
        ),  # (x, y, height, width)
        "enemies": Box(
            low=-np.inf,
            high=np.inf,
            shape=(self.max_enemies, 5),
            dtype=np.float32,
        ),  # (x, y, height, width, health)
    }
)
```

This observation space is quite complex, and determining the total number of observations required for the input layer in the DQN algorithm has become problematic for us.
One possible solution is to simplify the observation space to include only bot positions. This can be achieved as follows:

```python
self.observation_space = Box(
    low=np.array([0, 0, 0] * self.max_enemies, dtype=np.float32),
    high=np.array([1200, 700, 100] * self.max_enemies, dtype=np.float32),
    shape=(self.max_enemies * 3,),
    dtype=np.float32
)  # Implementation for 2 bots only
```

But this implementation is quite far from the best.
In my opinion, the best solution is to convert observations into an image format and use convolutional neural networks (CNNs) to process them.

3.1.3 Reset

The reset function initializes the simulation and returns the initial observation:

```python
def reset(self, *, seed=None, options=None):
    super().reset(seed=seed, options=options)
    self.simulation = Simulation()
    observation = self._get_obs()
    return observation, {}
```

3.1.4 Step

The step function controls player actions and advances the simulation:

```python
def step(self, action):
    move = action["move"]
    jump = action["jump"]
    shoot = action["shoot"]

    if move == 1:
        self.simulation.player.move(True)
    elif move == 2:
        self.simulation.player.move(False)
    if jump == 1:
        self.simulation.player.jump()
    if shoot == 1:
        self.simulation.player.shoot(0)   # Fixed-angle shooting

    self.simulation.next_step()
    observation = self._get_obs()
    reward = self._calculate_reward()
    done = self.simulation.game_over
    return observation, reward, done, False, {}

```

A key limitation of this approach is that the agent only controls the player, while the bots follow predefined behaviors. A better approach would be to allow AI to control bots, enabling self-play training.
Bot control could look like this (only works for bot class: PlayerLikeBot):

```python
 for bot in self.simulation.player_like_bots:
            if action == 1:
                bot.move(True)
            elif action ==2:
                bot.move(False)
            elif action ==3:
                bot.jump()
            elif action != 0:
                shoot_angle= (action-5)*10
                bot.shoot(shoot_angle)
```

3.1.5 Reward

The reward system encourages hitting enemies while penalizing getting hit or losing the game:

```python
def _calculate_reward(self):
    reward = 0
    if self.simulation.player.last_frame_bullet_hit:
        reward += 1
    if self.simulation.player.get_hit_last_frame:
        reward -= 2
    if self.simulation.player.last_frame_bullet_kill:
        reward += 10
    if self.simulation.game_over and self.simulation.player.health > 0:
        reward += 100
    if self.simulation.game_over and self.simulation.player.health <= 0:
        reward -= 100

    self.simulation.player.last_frame_bullet_hit = False
    self.simulation.player.get_hit_last_frame = False
    self.simulation.player.last_frame_bullet_kill = False
```

3.2 DQN

In this section, we present the architecture of the Deep Q-Network (DQN) model used to solve the reinforcement learning problem. DQN is a type of neural network designed to approximate the Q-value function, which is used to make decisions in environments where the agent interacts with the world in a sequential manner. The architecture consists of an input layer, one hidden layer, and an output layer.

```python
class DQN(nn.Module):
    def __init__(self, n_observations, n_actions):
        super(DQN, self).__init__()
        self.input_layer = nn.Linear(n_observations, 128)
        self.hidden_layer = nn.Linear(128, 128)
        self.output_layer = nn.Linear(128, n_actions)

    def forward(self, x):
        x = F.relu(self.input_layer(x))
        x = F.relu(self.hidden_layer(x))
        return self.output_layer(x)

```

Input layer
The model takes an observation (state) from the environment as input. The number of input neurons in the network (n_observations) is determined by the shape of the observation space. The input layer uses a fully connected (nn.Linear) layer, which connects each observation feature to 128 neurons.

Hidden layer
The model uses a fully connected hidden layer consisting of 128 neurons. This layer processes the input features and helps the network learn more complex patterns. The hidden layer uses the ReLU activation function, which introduces non-linearity to the network and allows it to approximate more complex relationships between the input and output.

Output layer
The output layer generates a vector with a number of neurons equal to the number of possible actions (n_actions). Each value in this output vector corresponds to the predicted Q-value for a specific action. The agent uses these Q-values to decide which action to take. The higher the Q-value for a particular action, the more desirable that action is.

Forward
The forward method defines the flow of data through the network. First, the input is passed through the input layer, followed by the hidden layer, with ReLU activations applied at each layer. Finally, the output layer is reached, which provides the Q-values for each possible action

4 Summary

The project involved developing a 2D platform shooter game in Python and attempting to train AI bots using reinforcement learning (RL). The game was fully implemented with mechanics such as character movement, shooting, different classes of weapons, collision detection, and different classes of randomly moving bots. Unfortunately we did not succeed in training bots to play effectively the game. However, the project provided a solid foundation for further work on AI integration and improving the training system.

References

[1] OpenAI Gym documentation. `https://www.gymlibrary.dev/index.html`.

[2] PyGame documentation. `https://www.pygame.org/docs/`.

[3] PyTorch documentation. `https://pytorch.org/docs/stable/index.html`.

[4] AI inspiration 1. `https://www.youtube.com/watch?v=kopoLzvh5jY`.

[5] AI inspiration 2. `https://www.youtube.com/watch?v=pkGa8ICQJS8`.

[6] Maxim Lapan. Deep Reinforcement Learning Hands-On - Second Edition, January 2020.