University of Milan

Department of computer science

Artificial Intelligence for Videogames project report

Author:
Diego Vallauri

Contents

1 Introduction and purpose of the project **2**

2 Creation of the environment **3**

3 First agent **5**

3.1 Environment set up. 5

3.2 Training set up 8

4 The training **11**

1 Introduction and purpose of the project

The project started with reference to the **Pong** game, where there are two players whose aim is to hit the ball without being overtaken by it. Starting from this, we decided the purpose of the project, which is to create an agent who can take the place of a player. In short, my goal was to create an agent who would hit the ball every time without missing it. To solve the problem, the first idea I proposed was a trigonometric solution. Since it wasn't that easy to put into practice, the Teacher suggested to use **Machine Learning** instead. This is the solution I adopted to the project and in particular I relied on **TensorFlow** which acts as an intermediary between Unity and Machine learning itself.
At the end of the project, I created three agents. The first moves only left and right (as in the **Pong** game), the second agent and the third in each four directions. Since the third was probably not required for the project, I consider it as an experiment I did towards the end.

2 Creation of the environment

Since the aesthetic part was not part of the project, I only used some Cubes and Cylinders to create the environment, as can be seen in the image below of the agent and the disk.

I have created two types of fields. One for training and one with two agents. I will explain the use of these fields in the next section.

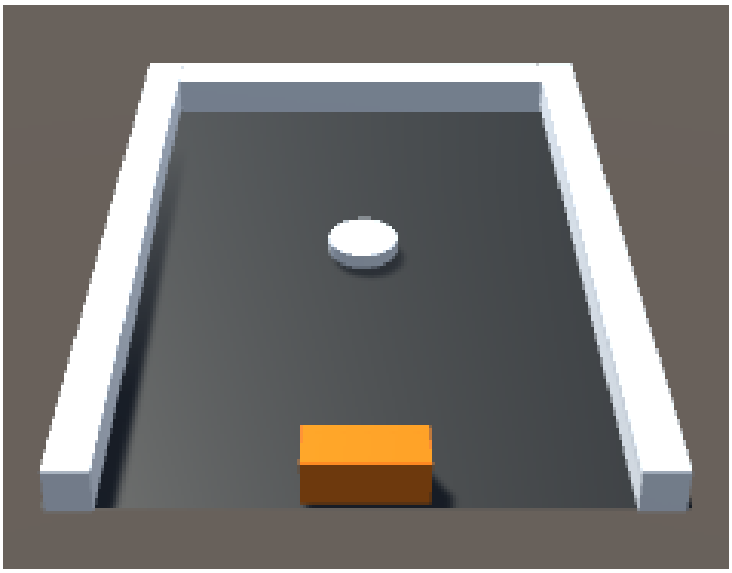

Figure 1: This is the field for training with just one side open.

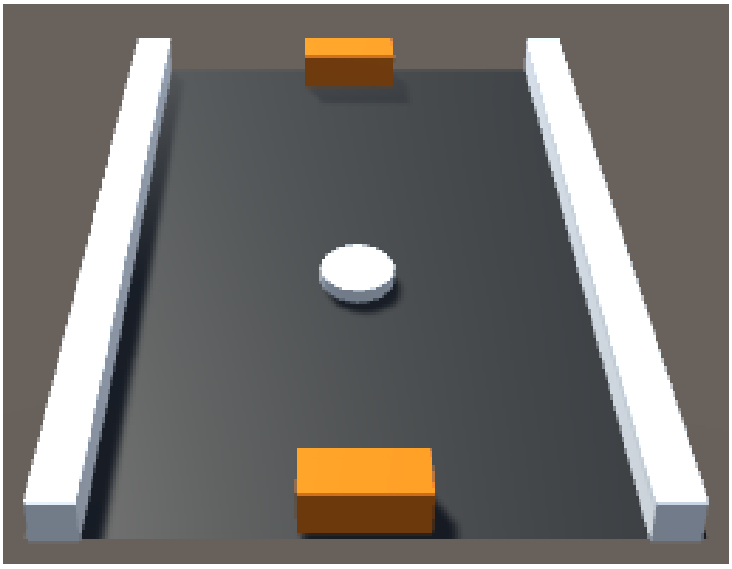

Figure 2: This is the normal field with two open sides.

3 First agent

The folder that describes the agent and the environment of this first example is located in **"Asset/FistTest"**.

3.1 Environment set up.

Before starting to train the agent, I obviously need to set my field. For the training I will use the *training field* (Figure 1). The first thought to decide is how to bounce the disk off the walls. Since every wall has a *Collider* and the disk a *RigidBody*, we can let the physics engine do it for us. Obviously I have to help it and have to make some considerations. For example, I don't want friction between the plane and the disk and between the disk and the wall. I also don't want the disk to stop moving after colliding with multiple walls. From this base, I created two different physical materials, each without friction but one with a bounciness equal to 1 and one equal to 0 (I don't want the disk to bounce on the plane). I assigned the first material to the side walls and the second to the floor.

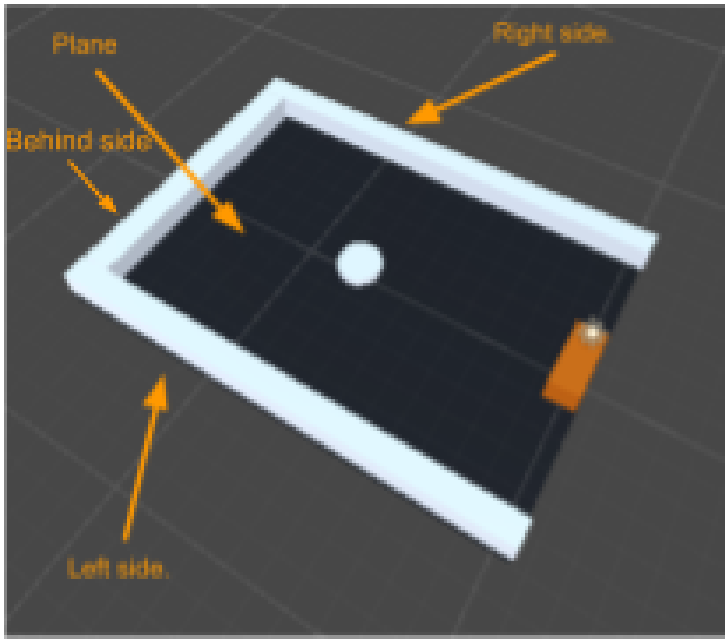

Figure 3: Representation of side walls and the plane.

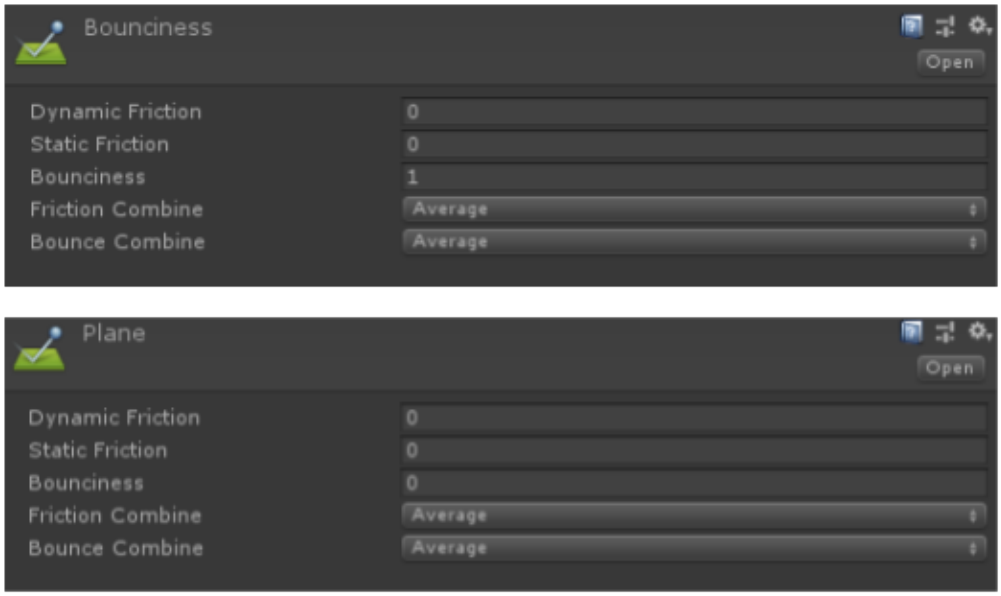

Figure 4: Physic material of the walls and the plane.

Now there is the "hardest" part: how do I make the disk bounce with the agent in a real way? First of all, I can't let the physics engine work because the disk will always behave the same way. I mean, if the ball goes straight through the middle to the agent, the ball will continue to change its Z coordinate but the X would remain the same (or at least, very similar. See figure 5) and in a nutshell its movement will remain the same. My solution is to look for the new destination and then add a force in that direction. For this, I used the Vector3.reflect function, which takes two parameters: in-direction and normal. In short I will calculate a new position that reflects the direction of the ball on the normal of the impact.

Figure 5: The only possible movements if the disk start straight to the agent.

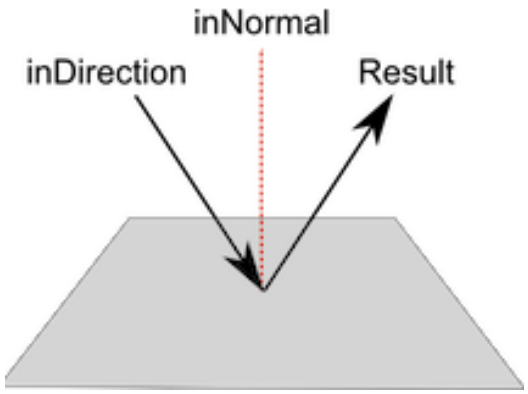

Figure 6: In summary, this is what the **Vector3.reflect** function returns.

So after find the new destination, I will apply a force based on new vector normalized and the speed of the disk. The code for this part is on the next page. As you will see, there is also an "OnTriggerEnter", whose sole purpose is to set the **haveILose** variable to true if the disk passes the agent (there is a game object in the scene called "onTriggerGoal" with the check mark on the "isTrigger" in the collider). The **haveILose** will be used on the agent for cheacking if the agent has lost and restart the episode, or not.

```
public class DiskBehaviour : MonoBehaviour{

    public float speed;

    private Rigidbody rg;
    private Vector3 destination;
    public bool haveILose = false;

    void Start(){
        rg = GetComponent<Rigidbody>();
    }

    private void OnCollisionEnter(Collision collision) {

        if (collision.gameObject.tag == "Agent") {

            ContactPoint contact = collision.contacts[0];

            rg.velocity = new Vector3(0f,0f,0f);
            Vector3 destination = Vector3.Reflect(transform.localPosition,
                                        collision.contacts[0].normal);
            rg.AddForce(destination.normalized * speed,
                        ForceMode.VelocityChange);
        }
    }

    private void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Goal")
            haveILose = true;
    }
```

Figure 7: Disk behaviour class.

One last thing I forgot to mention, is that this behaviour is attached to the agent but also to the behind wall (figure 3), since in training it simulates a real agent.

3.2 Training set up

Now i am ready to create my agent. As suggested, I create an empty game object as parent of my parallelepiped. Since i wanted to use a **RigidBody** for the physics and wanted to keep the agent part and the movement di-

vided, I simply assigned everything about the agent to the parent and the **RigidBody** and collider to the child.

- **Agent**: Has attached the **Behavior Parameters**, the **Decision Requester** and the **BAgent** (my script).
- **Player**: It is the son of **Agent** and has attached the **Box Collider**, the **RigidBody** and everything needed for rendering.

Let's now analyse the **BAgent** script. Since it extends the Agent class it has at least four methods: **Initialize**, **OnEpisodeBegin**, **CollectObservations**, **OnActionReceived**.

- **Initialize**: it has nothing special about it, just the initialization of the disk **RigidBody** and the agent child reference (that will actually move).

```
public class BAgent : Agent{
    [Header("Agent parameter")]

    public GameObject hockeyDisk;

    public float agentSpeed;

    private Rigidbody diskRB;
    private GameObject child;

    public override void Initialize() {
        diskRB = hockeyDisk.GetComponent<Rigidbody>();
        child = transform.GetChild(0).gameObject;
    }
```

- **OnEpisodeBegin**: I first take the value of **haveILose** and set it to false, since the true value is the trigger event for the end of the episode. So I create a different position for the agent, for the disk and calculate a new starting direction for the disk. At the end I add a force in the calculated direction.

```
public override void OnEpisodeBegin() {
    hockeyDisk.GetComponent<DiskBehaviour>().haveILose = false;
    hockeyDisk.transform.localPosition = createRandomPosition(false,false);
    float speed = hockeyDisk.GetComponent<DiskBehaviour>().speed;
    child.transform.localPosition = createRandomPosition(false, true);
    Vector3 pos = createRandomPosition(true,false);
    diskRB.velocity = new Vector3(0f, 0f, 0f);
    diskRB.AddForce(pos.normalized * speed,
                    ForceMode.VelocityChange);
}
```

The **createRandomPosition** method, is only an auxiliary method to create the position of the elements based on the parameters. (It's not that interesting so I don't put any photos there, but it's obviously present in the folder).

- **CollectObservations**: this is quite important and contains seven observations. The first three are the component of the local position of the disk, The fourth is the x local position of the agent and the last three are the component of the RigidBody velocity.
  In this part I made some mistake at the beginning, for example I gave the global and not the local position of the elements and this create problems during the training because every prefabs has it's own global position and so the learning didn't go very well. Another mistake (not really a mistake) was providing the agent y and z components as well (which are fixed) and this only slowed down the training.

```
public override void CollectObservations(VectorSensor sensor) {
    sensor.AddObservation(hockeyDisk.transform.localPosition);
    sensor.AddObservation(child.transform.localPosition.x);
    sensor.AddObservation(diskRB.velocity);
}
```

- **OnActionReceived**: the agent will receive only a float (clamped beetween -1 and 1) and will first reset the RigidBody velocity of the agent and then add a force in the direction received before.
  Moreover the agent will receive a positive reward of 0.05 if the disk has not overcome it or a negative reward of -5 in the opposite case.

```
public override void OnActionReceived(float[] vectorAction) {
    Vector3 destination = new Vector3(Mathf.Clamp(vectorAction[0], -1f, 1f),
        child.transform.localPosition.y, child.transform.localPosition.z);

    child.GetComponent<Rigidbody>().velocity = new Vector3(0f,0f,0f);
    child.GetComponent<Rigidbody>().AddForce(destination.normalized * agentSpeed,
            ForceMode.VelocityChange);

    if (hockeyDisk.GetComponent<DiskBehaviour>().haveILose == false) {
        SetReward(0.05f);
    } else {
        SetReward(-5f);
        EndEpisode();
    }
}
```

4 The training

For the learning i will use the scene **Test**.

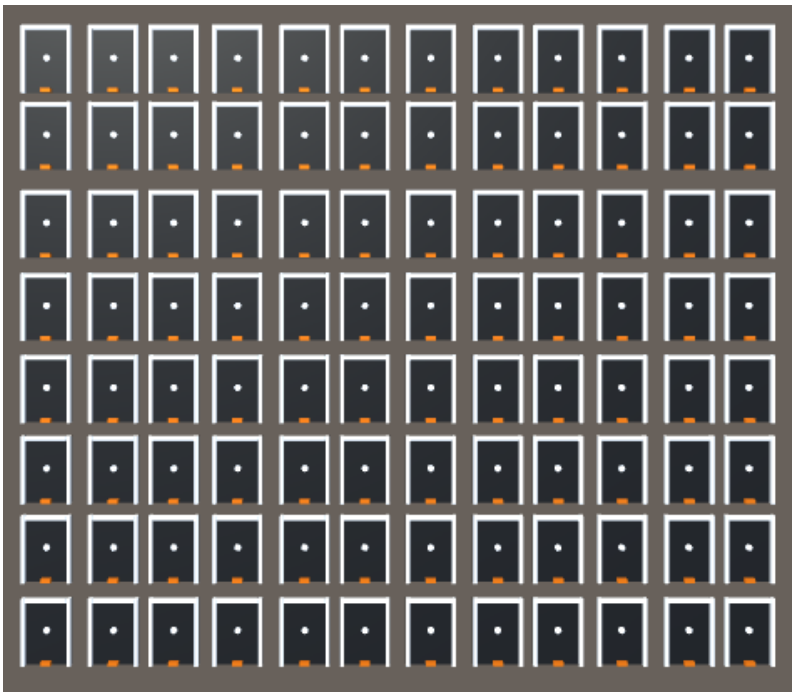

Figure 8: Scene Test with 96 field prefabs.

For the .yaml file, I followed the instruction on the doc site. In particular in this first example the .yaml file is composed in that way:

```yaml
behaviors:
  Field:
    trainer_type: ppo
    hyperparameters:
      batch_size: 64
      buffer_size: 12000
      learning_rate: 0.0002
      beta: 0.001
      epsilon: 0.2
      lambd: 0.95
      num_epoch: 3
      learning_rate_schedule: linear
    network_settings:
      normalize: true
      hidden_units: 128
      num_layers: 2
      vis_encode_type: simple
      memory: null
    reward_signals:
      extrinsic:
        gamma: 0.99
        strength: 1.0
    init_path: null
    keep_checkpoints: 5
    checkpoint_interval: 1000000
    max_steps: 1000000
    time_horizon: 1000
    summary_freq: 100000
    threaded: true
```

Figure 9: Behaviour.yaml

As can be seen from above, the learning lasted 1000000 steps.
The learning works very well and the agent is able to hit the ball every time (Look at the "Fist test training" video in the folder). After the first training i went in panic since i tried to put tu player against. But in this case the one

in the opposite position of the training, was not able to catch the ball. First I thought it was my error, but after some trial a think the problem it's just the refernce system. Before