MTEAC Operation Manual

This paragraph is an introduction on how to understand and use MTEAC by the order of, basic concept, function, method and code structure.

In the paragraph of introduction about basic concepts, we will not involve anything about concrete coding and code structure. The method we used to achieve our expectation will be introduced later, then we will introduce how we turn the concept of our description of the world project to specific code.

The purpose of this paragraph is to explain the basic logic behind MTEAC and it is not going to refer to the detailed content of “world-project”. The content of the specific world project will be introduced in other documents.

Table of Contents

Basic concept and structure······3

Function······11

Method of implementation······12

Code structure······17

How to customise “world_project”······18

Adaptation of openAI gym······19

Other instructions······21

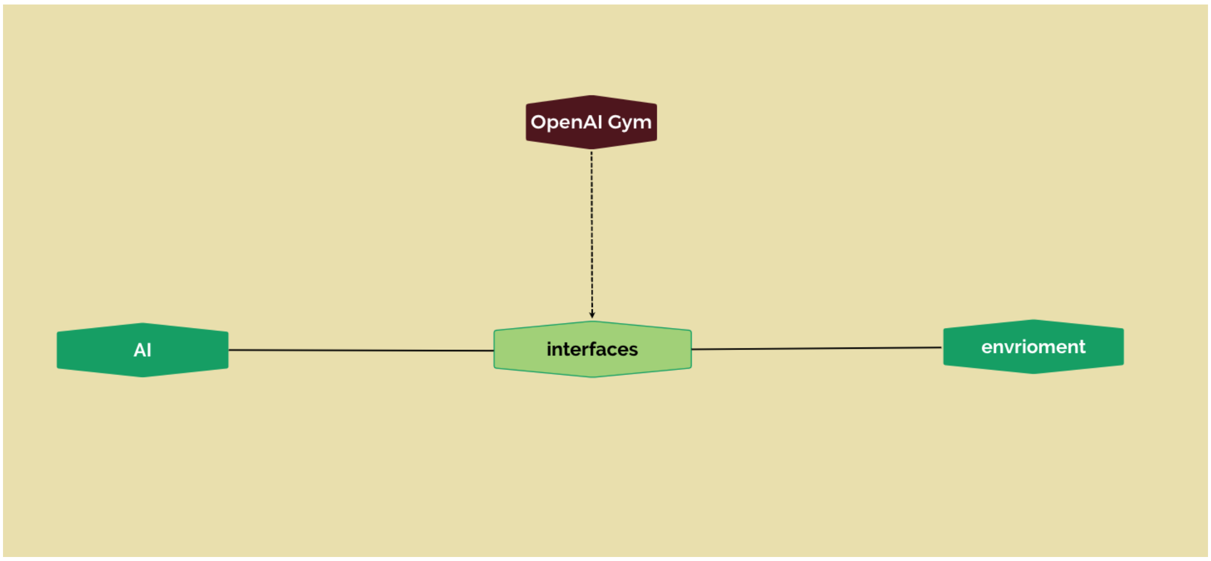
Extra······24

The basic structure and concept

1. Basic concept

Environment:

In our understanding, the concept of “ Environment” is: a describable status that exists in a specific particular point in time, it can provide action space and state to the AI agent and it can receive orders and instructions and it can change itself over time.

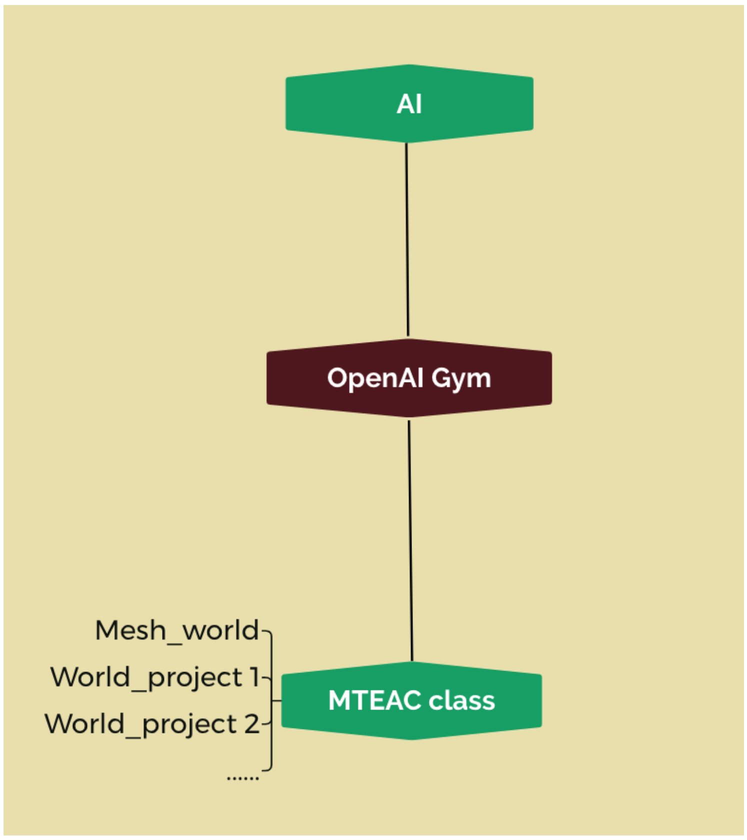


The MTEAC class:

This class adapted with OpenAI gym so it is an environment class that interacts with OpenAI’s agent. In MTEAC, WorldEnv represents the environment to openAI but it is not the environment itself, it is a “Management tool of the environment”. This class will chose a specific environment as the environment of OpenAI Gym with factory pattern in design pattern, and we call this specific environment, the" world project”

World Project:

Each specific environment is called a world project; When MTEAC uses a different world project, the “WorldEnv " will be seen as a different environment.



2. Structure of world project

The World project mainly included the description , the generate method and change logic of status.

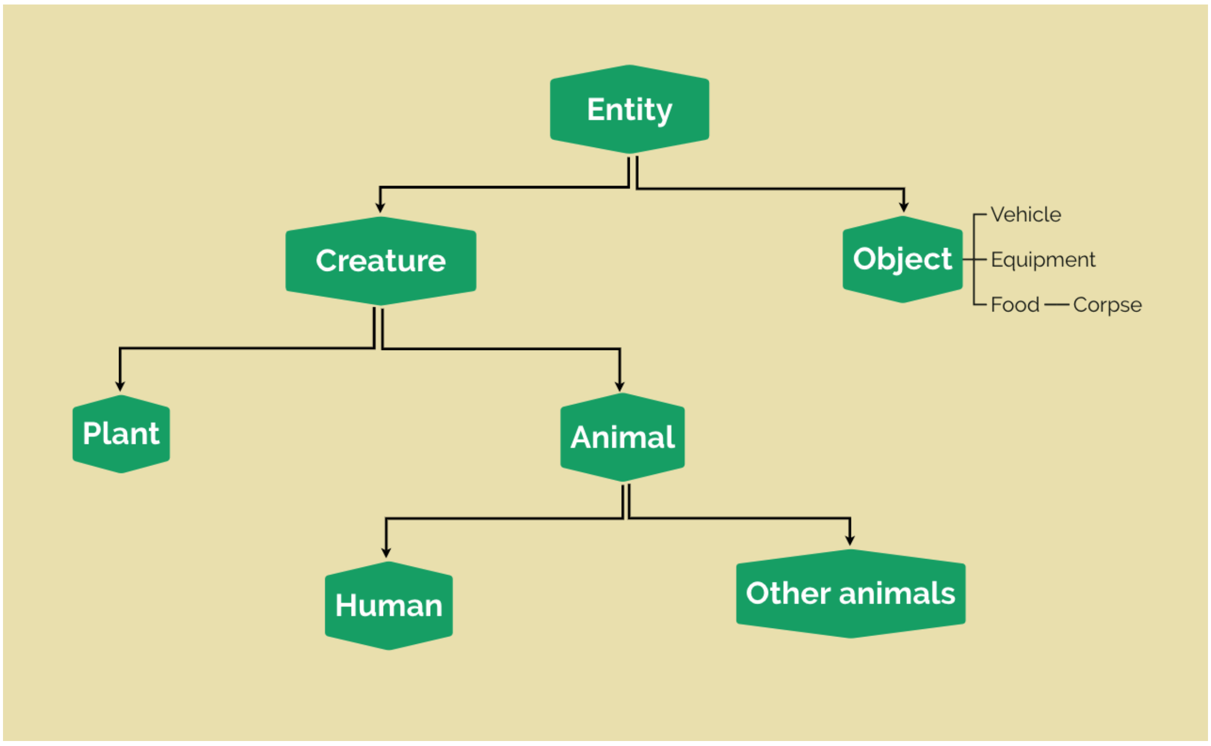
In MTEAC, we believe “ Status” should be description by these three basic elements, “spatial properties”,”Entity” and “Environment variable” “spatial properties”: for example, the terrain, weather and other elements that are defined by different locations in space. So we called them “spatial properties”

“Entity”: Everything can be interacted with, for example, animals and tools.
“Environment variable”:Everything that has a global influence, for example: time, seasons and change between day and night.

We use two-dimensional arrays to represent the “spatial properties” and use simple variables to represent “environment variables”; But more complicated on “Entity”, we represent different entities by Classifying them and letting them inherit different classes.
For example, all entities can be classified into “creature” and “non-creature”(“obj”), and creatures also can be classified into “plants” and “animals”.
We define these static classes and give them an inheritance method, we call this “Entity Structure”.

3. The default Entity Structure

In MTEAC, we defined a default Entity Structure(which is still being optimising)



In different world projects, All entities class will inherit from the default entity structure; For example, if we would like to define"Monkey" and make it exist in our world project, then we could define " Monkey" class and make it inherit from the "animals" class from the default structure.

Each different class of entities has different attributes and logic. For example, all animals have the ability to "move" and an attribute of "Speed", but plants have none of them.

We could describe the different concepts under the default structure with this.

Entity:

·Attribute: Position, id

·Functions: move, post_ture_chage(here we do not list specific functions like "get_id")

·Explanation:

A static entity is an actual being in the world, each of them has a location. In MTEAC, we have an unique ID for each entity for convenience to manage.

The “ post_ture_chage” is the change that happens to entities when each round passes, for example: trees grow taller, people get hungry. This method is called once at the end of each turn.

Creature:

·Attribute: Health Point, carapace

·Functions: die, be_attack

·Explanation: Creature is such a thing which has two status, “live” and “dead”, and they can be hurt by other things. They can also breed themselves, but we have only achieved the function of plant reproduction in the “mesh_world”, Also we were planning on simulating a gene system but did not manage to achieve due to the limited period of time.

Obj(non-living things):

·Attribute: None

·Functions: None

·Explanation: we have not add any specific attribute to the static class of “Obj”, we were planning on adding things like “ weight” ,”size” or”shape” but did not achieved this due to the limited period of time.

Animals:

·Attribute: action_list, other more detail attributes will be defined in specific world_project

·Functions: judge_action_validity, action_cost, action_interior_outcome, body_change, get_perception

·Explanation:

The main difference between animals and plant objects is that they can act. Animals are active, so animal’s specific methods are methods relating to motion.

1. judge_action_validity: Judge the legitimacy of the behaviour according to the action instructions and the current state of the animal.
2. action_cost : To get the cost of specific action.
3. action_interior_outcome: the outcome of action, for example: eating will make animals less hungry.
4. body_change: functions that interact will body state.
5. get_perception: input a State instance and put the results of perception.

Plants :

·Attribute: None

·Functions: None

·Explanation:

we did not set specific attributes to plants, because most of them are logically just like objects.

We do not achieve the breeding function of plants in the default entity structure.

Human:

·Attribute:None

·Functions: None

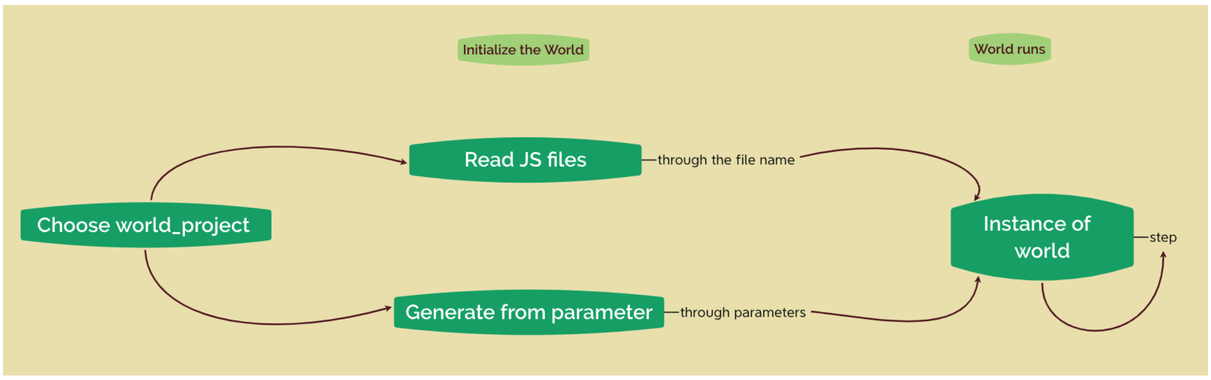
·Explanation:

No special attributes were added to human class in the default entity structure. But in the “mesh_world”, humans have great differences with animals, they can use tools or craft new objects.

The reason we did not add any special attributes is because we hope users are able to customise the differences between humans and animals. These settings could be further optimised.

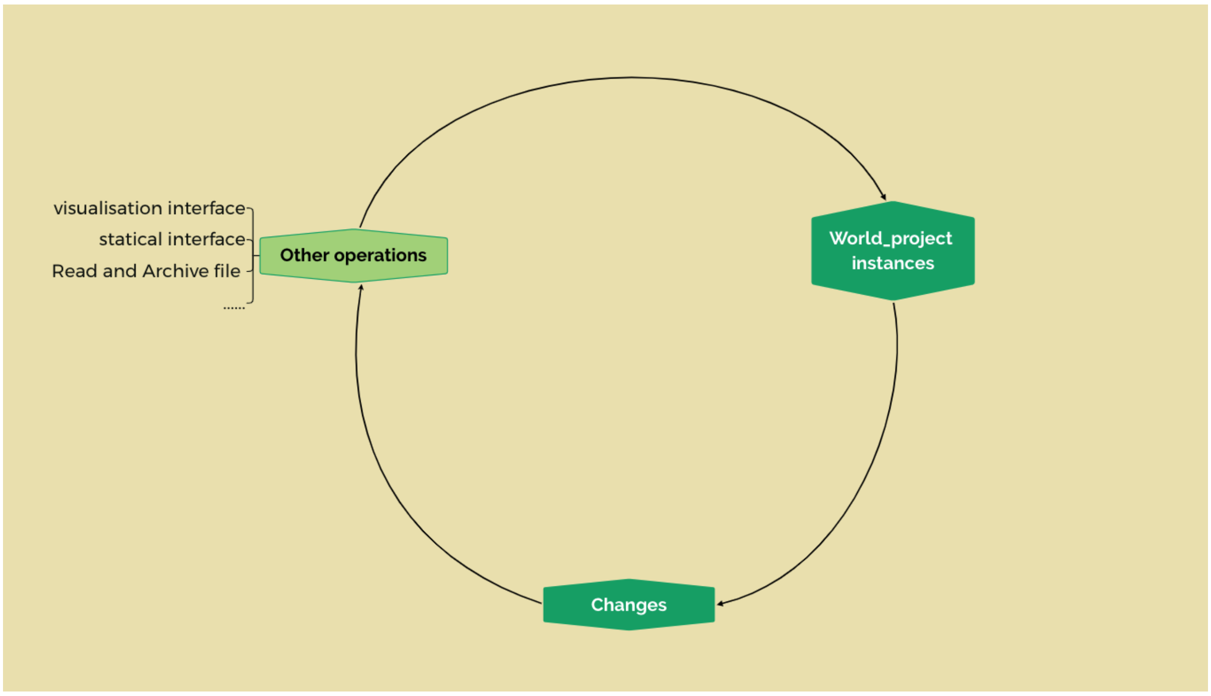
The default entity structure was there for the convenience to define specific entity structure in specific world projects. Each world_project could define their own entity structure, and does not have to follow the default structure.

The operation of MTEAC

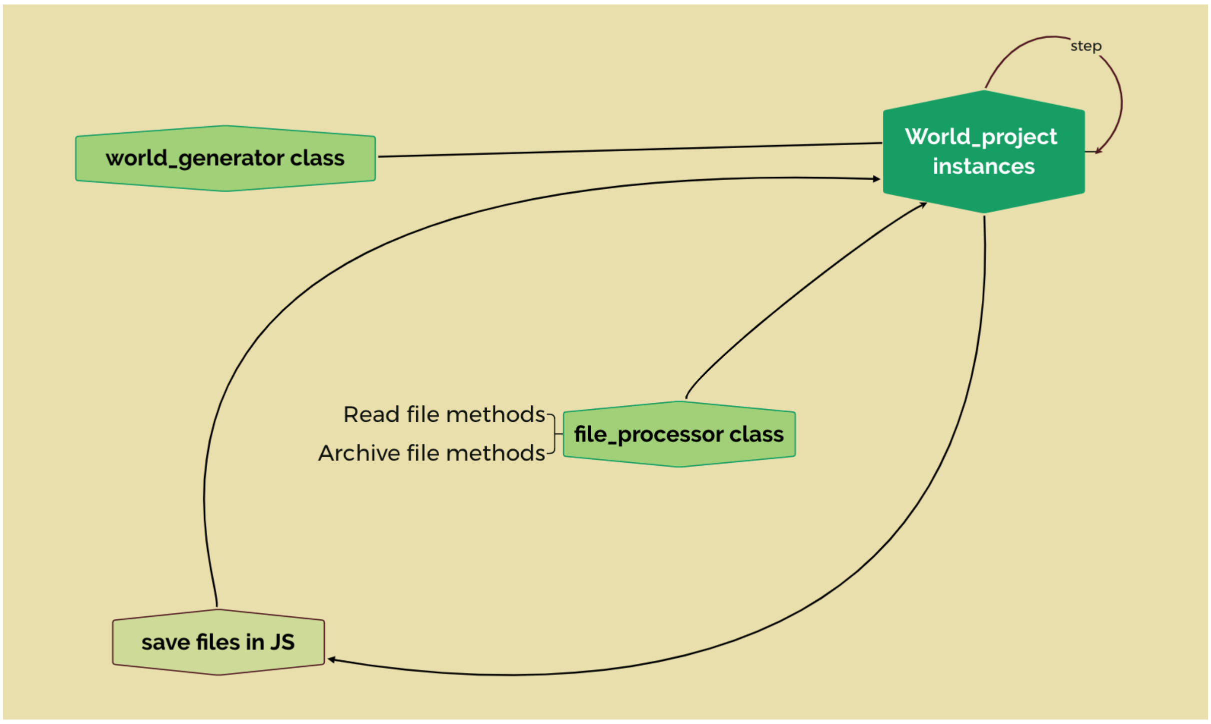


The MTEAC class will initialise a world according to the selected “World_Project” first.

Then generate an instance of the world of that world project and run it. As time passes, the world will change from one state to another state.



After the step() is executed, users can add other user-defined methods, for example, visualisation interface, statical interface, read and archive file etc.



For example, during the world instance running, after execution of step(), the user can call the archive and load method of File_processor to archive or load the world instance.

For example during the world instance operation, The world instance can be archived and read by calling File_processor's archive and read methods after any step() operation.

Functions

We intended to create a graphical interface to manage the following functions but were unable to do so due to time and ability. Therefore, the following description of functions need to be implemented by modifying the code.

This page is just a list of the things that you can do and the next page is how do you do them

1. World_project management

    A. create world_project

    B. Let MTEAC select a world_project

    C. Modify/customize the World Project

2. Use default archive/reader to archive/reader a world instance

3. Unimportant features

    A. Background functions

    B. Statistics function

The implementation method of the function

File structure:

main.py is the program entry (for test). WorldEnv in world_env_interface.py is the MTEAC class.

The ".py" files under the world folder are all abstract classes, which is for specification for world projects.

All world project folders are stored in the world_project folder.

The default entity structure is stored in the Entity folder under the world folder.

1. world_project management

A. create world_project

To create a “world_project”, users simply create a folder under world_project and after that users don't have to modify or register anywhere because the MTEAC class automatically reads the folder path.

We provide a more convenient way to create world_project: just copy the blank_world folder and change the “blank” in the folder name to whatever name you want and then change all the “blank” and “Blank” in the files inside to that name as well (remember to use uppercase for class names).
It is recommended to use the character replacement function of the document editor and be case sensitive.

If want create a new world project by create a new folder, please follow the following format (this format is being improved) :

· Folder names must end with "_world"

The part before "_world" is the name of the world_project that will be used in building the world_project later on.

There must be the following files in the folder:

· exhibitor.py

This file must have an exhibitor class and must inherit the Exhibitor_super abstract class from exhibitor_super.py under the world file folder.

· state.py

The file must have a class named *world_project name*+ "_state". For example, the class in blank_world state.py is called Blank_state.

This class must inherit from the State abstract class in state.py under the world file folder.

· world.py

The file must have a class named *world_project name*+ "_world". For example, the class in blank_world state. Py is called blank_world.

This class must inherit from the World abstract class in world.py under the world file folder.

· world_generator.py

There must be a class called "Concrete_world_generator" in this file

This class must inherit from the abstract world_generator class in world_generator.py under the world file folder.

· A save folder which just has an empty folder called Save.

(if you want to use the default archive/reader functions)

·The class under world.py must have the "get_openAI_action_space_and_observation_space" method to specify the "observation_space" and "action_space" required by OpenAI.

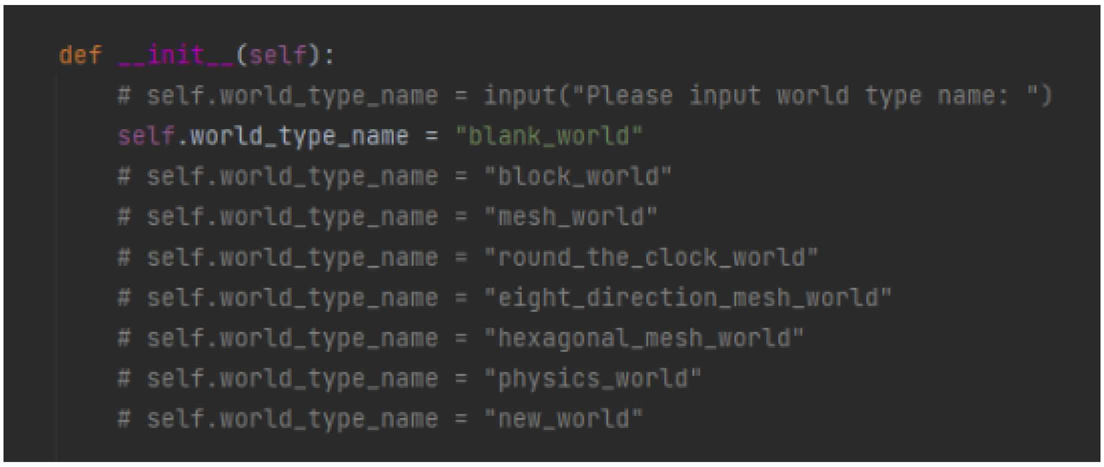
This method returns a tuple with a format of (action_space, observation_space). Both elements must be space objects specified by openAI GYM.

· mods.py

The files are set to make the import State and World classes more convenient. There must be one mods.py file if the default reader is to be used.

B. let the MTEAC class point world_project

If you want the MTEAC class to select a world_project, change the self.world_type_name variable in the __init__() method of the MTEAC class (WorldEnv class under world_env_interface.py) to the same as the name of the world project file folder (A string here).



```
def __init__(self):
    # self.world_type_name = input("Please input world type name: ")
    self.world_type_name = "blank_world"
    # self.world_type_name = "block_world"
    # self.world_type_name = "mesh_world"
    # self.world_type_name = "round_the_clock_world"
    # self.world_type_name = "eight_direction_mesh_world"
    # self.world_type_name = "hexagonal_mesh_world"
    # self.world_type_name = "physics_world"
    # self.world_type_name = "new_world"
```

C. Modify/customise the world project

See below [Custom methods]

2. Use the default archive/load method

All we need to do is call the archive() or load() methods under file_processor.py wherever we want to read the file. In the default archive/reader, both of these methods have been decoupled from the specific world project, so for all world projects it will work, but the default archive and reader may be less efficient due to its generality and json file structure.

Archive () reads three parameters: “state”, “world_type_name”, and “file_name”.

1. “state” is the state attribute of the current world that represents the state that needs to be archived at this stage.

2. “world_type_name” is a string to represent the name of the world project.

3. “file_name” is the name of the archive file.

archive file: After calling this archive() method, a .save file named “file_name” will appear under the folder of the corresponding world project folder, which is the archive file.

The load() method reads two parameters, “world_type_name” and “file_name”.

1. “world_type_name” uses the same string as the name of the world project (also the same string as the string used in the archive() method).

2. “file_name” is the name of the archive file that the load() method reads to generate a corresponding state instance.

In our project, we have not implemented the reset() method required by openAI, so we temporarily use the file reading method instead, which is another point to be optimised.

3. Other features

A. Background function

B. Statistics function

We implemented a simple background and statistics function but it didn't work very well so we won't talk about it here and we will talk about it in detail after optimization.

Code Structure

Correspondence between concept and code:

Under the world folder, the world.py and state.py both are abstract classes that specify the World and State class. The comments under these two files explain the methods needed to build them and the meaning of each method.

Execution order: Take the example of OpenAi gym in main.py as an example

Initialization: env = WorldEnv()

Operation: env.step()

When step() is called:

step() in World_env_interface.py will be called. Then in step(), the **take_action()** and **evolution()** methods of the corresponding world.py in the world project will be called. These two methods need to be read in an action **command** parameter and an **id** parameter. The id is used to specify which entity is the subject of the instruction to be executed. take_action() causes the specified entity to execute the command once. **evolution()** causes the environment to change over time once and other entities to operate on their own once.

The method to customise the world project

We divide the general process of customising the world project into the following steps:

1.Describe the state of the world concretely and determine:

This step determines the description of the world at the conceptual level.

What are the properties of the world? What data structure is used to represent each of these properties?

How do the properties interact with each other?

How do the properties change?

Whether the properties change over time, based on behaviour, or based on other properties?

How are the properties generated when the world is initialised?

2.Conceptualize how to implement these properties and do the following:

Find the location of the file corresponding to the concept.

Adding properties and logic.

Adaptation of openAI gym

To adapt MTEAC into openAI gym, you only need to follow these steps:

1. Drag the world folder and the world_env_interface.py folder from the MTEAC directory into the site-packages\gym\envs\classic_control folder.

Note: mesh_world uses some C++ code. So, to use mesh_world, you need to drag the C++ folder into that folder as well.

2. In the _init_.py file in the gym\envs directory, add the following code:

register(

```
        id="MTEAC-v0",
  entry_point="gym.envs.classic_control:WorldEnv",
        max_episode_steps=200,
        reward_threshold=100,
)
```

3. Add the following line of code to the __init__.py file in the gym\envs\classic_control directory:

```
from gym.envs.classic_control.world_env_interface import WorldEnv
```

After completing the above steps, you can call the environment we created ourselves.

Create a new main.py file, and then write the following code to call the environment:

```
import gym


# program entry


if __name__ == "__main__":


    ai_mode = 1
    ai_num = 2


    env = gym.make("MTEAC-v0")
    env.set_ai_num(ai_num)
    env.reset()


    """
```

```
        openAI mode
    """
    while True:
        # Take a random action
        action = [env.action_space.sample() for num in range(ai_num)]
        obs, reward, done, info = env.step(action)

        if done[0] == 1:
            print("pac man win")
            break
        elif done[0] == -1:
            print("pac man lost")
            break

        # Render the game
        env.render("ai")

        if done is True:
            break

    env.close()
```

Other instructions

About comments

We have added comments to MTEAC's abstract class to illustrate the concept of methods and classes.

About demonstration of implementation of world project

We will place a document in the root directory to demonstrate the implementation of a simple world project.

There is also a demonstration document in the folder of blank_world, block_wolrd and mesh_world also.

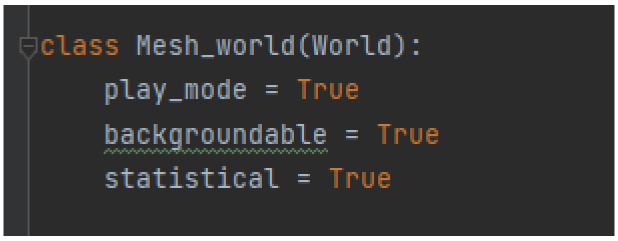
The format of the instruction about openAI gym

Due to the support for multi-agent, the step() method of WorldEnv will need to read an array of many commands rather than a single command. The index of the command in the array corresponds to the ID of the entity. For example, env.step(["eat", "run"]) means that the entity with id 1 performs the eat action, and the object with id 2 performs the run action and so on.

ai_mode and game_mode in main.py

When ai_mode = 1, main() will execute WorldEnv by openAI mode. When ai_mode = 2, main() will execute WorldEnv by game mode.

Game mode refers to user control of an entity by keyboard. openAI mode refers to all entities controlled by AI. Please note that game mode is not available for all world projects. If you want to control an entity manually, you need to add class variable “play_mode = True” into the class Xxxx_world in the world.py in your world project folder, and program a method that makes the world load the user's input in display() in exhibitor.py. Users can refer to block_world, in it users can control a block to move and push other blocks in this sample world.



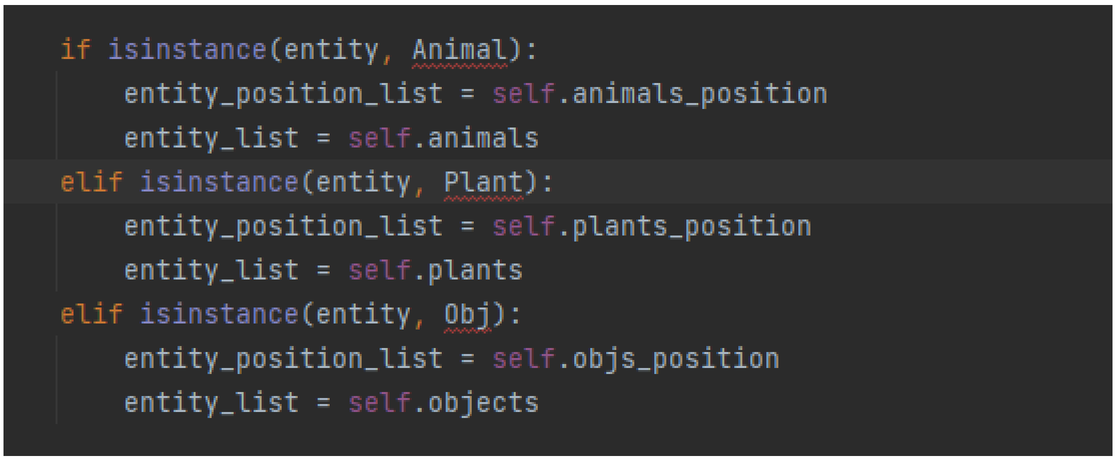
```
class Mesh_world(World):
    play_mode = True
    backgroundable = True
    statistical = True
```

About entity\entity_import.py and world_project\XXXX_world\entity\entities.py

The program will read all .py files that are in the same folder with entity_import.py (sub folder will also be read), and then automatically import classes that are mentioned by all .py files. So please follow the naming convention of folder “entity”, all class modules under the folder “entity” should have the same name as the file with the first letter capitalised for avoiding the automatically importing module not being able to read the class, such as class name in human.py in world\entity\creature\animal\human should be “Human”.

The code of make .py file in the other position to load the module in world/entity is “from world.entity.entity_import import *”.

If you meet this kind of warning, please ignore it, this is because the IDE cannot understand our automatically importing module.



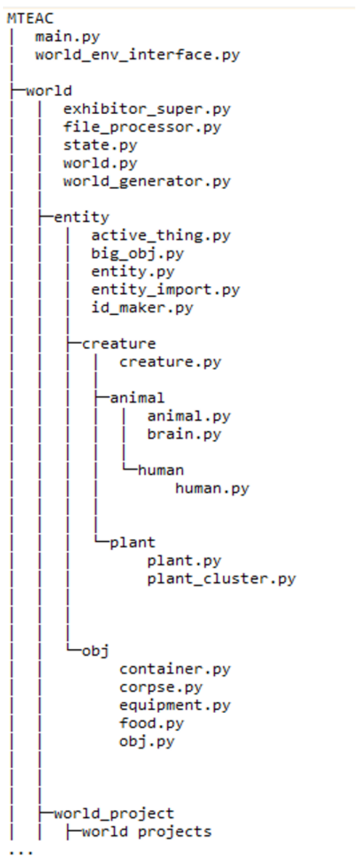
```
if isinstance(entity, Animal):
    entity_position_list = self.animals_position
    entity_list = self.animals
elif isinstance(entity, Plant):
    entity_position_list = self.plants_position
    entity_list = self.plants
elif isinstance(entity, Obj):
    entity_position_list = self.objs_position
    entity_list = self.objects
```

Incomplete parts

Due to time constraints, the project still has a large number of parts that need to be improved. In the future, we will revise and improve these places. (we will list what needs to be improved later)

Appendix

Document Architecture:



```
MTEAC
│   main.py
│   world_env_interface.py
│
├─world
│   │   exhibitor_super.py
│   │   file_processor.py
│   │   state.py
│   │   world.py
│   │   world_generator.py
│   │
│   ├─entity
│   │   │   active_thing.py
│   │   │   big_obj.py
│   │   │   entity.py
│   │   │   entity_import.py
│   │   │   id_maker.py
│   │   │
│   │   ├─creature
│   │   │   │   creature.py
│   │   │   │
│   │   │   ├─animal
│   │   │   │   │   animal.py
│   │   │   │   │   brain.py
│   │   │   │   │
│   │   │   │   └─human
│   │   │   │           human.py
│   │   │   │
│   │   │   └─plant
│   │   │           plant.py
│   │   │           plant_cluster.py
│   │   │
│   │   │
│   │   └─obj
│   │           container.py
│   │           corpse.py
│   │           equipment.py
│   │           food.py
│   │           obj.py
│   │
│   │
│   ├─world_project
│   │   ├─world projects
...
```

Requirement of World_project

Please follow the formats below if you want to create a new world project:

1. Create a new folder in world/world_project. Folder name must be "XXXX_world", XXXX is the name of world_project, XXXX can also be used to construct world_project.

2. Folder must contain these files:

    a) exhibitor.py. It must have Class Exhibitor, and must inherit the abstract class Exhibitor_super from world/exhibitor_super.py.

b) state.py. It must have a class named “XXXX_state”, XXXX is the name of world_project. For example, in blank_world, the name of the class in state.py is Blank_state. It also must inherit the abstract class State from world/state.py

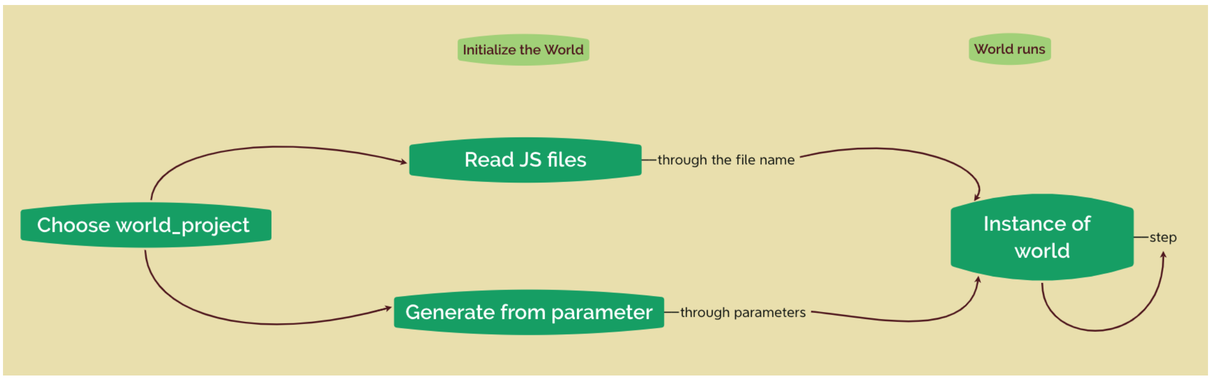
c) world.py. It must have a class named “XXXX_world”, XXXX is the name of world_project. For example, in blank_world, the name of class in world.py is Blank_world .It also must inherit the abstract class World from world/world.py

d) world_generator.py. It must have a class named Concrete_world_generator. Concrete_world_generator must inherit abstract class World_generator form world/ World_generator.py

e) If you want to use the default saver/loader, you need to create a folder named save.

f) world.py. It must have methods get_openai_action_space_and_observation_space to return a tuple that involved two elements that OpenAI needed: observation_shape and action_space, these two elements must be space that openAI gym ruled.

The running order of the whole program:


Initialize the World
World runs
Read JS files
through the file name
Choose world_project
Instance of world
step
Generate from parameter
through parameters