gobigger

Release 0.2.0

OpenDILab Contributors

Jan 29, 2023

TUTORIAL

CHAPTER

ONE

OVERVIEW

GoBigger is a **multi-agent** environment for reinforce learning. It is similar to Agar, which is a massively multiplayer online action game created by Brazilian developer Matheus Valadares. In GoBigger, players control one or more circular balls in a map. The goal is to gain as much size as possible by eating food balls and other balls smaller than the player's balls while avoiding larger ones which can eat the player's balls. Each player starts with one ball, but players can split a ball into two once it reaches a sufficient size, allowing them to control multiple balls.

GoBigger allows users to interact with the multi-agent environment easily. Through the given interface, users can get observations by actions created by their policy. Users can also customize their game environments by changing the args in the config.

CHAPTER

TWO

INDICES AND TABLES

2.1 Installation

2.1.1 Prerequisites

System version:

- Centos 7
- Windows 10
- MacOS

Python version: 3.6.8

2.1.2 Get and install GoBigger

You can simply install GoBigger from PyPI with the following command:

```
pip install gobigger
```

If you use Anaconda or Miniconda, you can install GoBigger through the following command:

```
conda install -c opendilab gobigger
```

You can also install with newest version through GitHub. First get and download the official repository with the following command line.

```
git clone https://github.com/opendilab/GoBigger.git
```

Then you can install from source:

```
# install for use
# Note: use `--user` option to install the related packages in the user own directory(e.g.
↪: ~/.local)
pip install . --user

# install for development(if you want to modify GoBigger)
pip install -e . --user
```

2.2 Quick Start

2.2.1 Launch a game environment

After installation, you can launch your game environment easily according the following code:

```
import random
from gobigger.envs import create_env

env = create_env('st_t2p2')
obs = env.reset()
for i in range(1000):
    actions = {0: [random.uniform(-1, 1), random.uniform(-1, 1), 0],
               1: [random.uniform(-1, 1), random.uniform(-1, 1), 0],
               2: [random.uniform(-1, 1), random.uniform(-1, 1), 0],
               3: [random.uniform(-1, 1), random.uniform(-1, 1), 0]}
    obs, rew, done, info = env.step(actions)
    print('[{}] leaderboard={}'.format(i, obs[0]['leaderboard']))
    if done:
        print('finish game!')
        break
env.close()
```

You will see output as following. It shows the frame number and the leaderboard per frame.

```
[0] leaderboard={0: 3000, 1: 3100.0}
[1] leaderboard={0: 3000, 1: 3100.0}
[2] leaderboard={0: 3000, 1: 3100.0}
[3] leaderboard={0: 3000, 1: 3100.0}
[4] leaderboard={0: 3000, 1: 3100.0}
[5] leaderboard={0: 3000, 1: 3100.0}
[6] leaderboard={0: 3000, 1: 3100.0}
[7] leaderboard={0: 3000, 1: 3100.0}
[8] leaderboard={0: 3000, 1: 3100.0}
[9] leaderboard={0: 3000, 1: 3100.0}
[10] leaderboard={0: 3000, 1: 3100.0}
...
```

2.2.2 Customize your config

Users can also choose to customize the game environment by modifying the configuration cfg and through the `gobigger.envs.create_env_custom` method we provide. The `gobigger.envs.create_env_custom` method accepts two parameters, the first parameter is `type`, the optional value is `st` or `sp`, which represent the standard game mode, and the independent action game mode. See the introduction of the two modes for details. Below we give a few simple examples based on the standard game mode.

Add more players in a game

For example, you may want to allow 6 teams and 2 players per team in your game, and then please modify `team_num` and `player_num_per_team` in config.

```python
from gobigger.envs import create_env_custom

env = create_env_custom(type='st', cfg=dict(
    team_num=6,
    player_num_per_team=2
))
```

Extend the game time

If you want to extend the game time to 20 minutes (24000 frames), you can use the following codes.

```python
from gobigger.envs import create_env_custom

env = create_env_custom(type='st', cfg=dict(
    frame_limit=24000
))
```

Change the size of the map

If you want to have a larger map, you can change `map_width` and `map_height` in config.

```python
from gobigger.envs import create_env_custom

env = create_env_custom(type='st', cfg=dict(
    map_width=1000,
    map_height=1000,
))
```

2.3 What is GoBigger?

2.3.1 Overview

GoBigger is an Agar like game, which is one of the most popular game all over the world. In GoBigger, users control their owned balls to collect weight in a 2-d map with border. There are many other balls in the map, such as food balls, thorns balls, spore balls and other players' balls. With the limited time, user are ranked by the weight they collect. In order to improve the antagonism of the game, users are allowed to eat other others' balls to get more weight. So in a game, users need to grow by eating food-balls and thorns-balls, and escape from the larger balls, and eat smaller balls to gather more weight quickly.

For a more detailed exposition rules of the game, we will firstly introduce the balls in GoBigger.

2.3.2 Balls

- Food balls

  Food balls are the neutral resources in the game. They will stay on where they are borned util someone comes and eats them. If a clone ball eat a food ball, the food ball's size will be parsed to the clone ball. There is a maximum number of food balls in the map. Once the number of available food balls is not enough, the game will generate new food balls in random positions every certain time.

- Thorn balls

  Thorns balls are also the neutral resources in the game. Different with food balls, they have larger size and less quantity. If a clone ball eat a thorn ball, the thorn ball's size will be parsed to the clone ball. But at the same time, the clone ball will explode and will be splited into several pieces. In GoBigger, the pieces always have the same size, and they will appear evenly around the original clone ball. So be careful! If there are other clone balls larger than yours, you should pay more attention to decide whether to eat the thorns-balls.

  Another feature of the thorns-balls is that they can be moved by clone balls through ejecting spore balls. If a clone ball eject spore balls to a thorn ball, the thorn ball will eat the spore balls and grow larger. Besides, it will move on the moving direction of the spore balls. That means if a clone ball wants to force a larger clone ball to explode, it can eject spore balls to the thorn ball and move it to the larger clone ball.

- Spore balls

  Spore balls are ejected by the clone balls. They will stay on the map and can be eaten by any other clone balls. Spore balls can not move or eat others.

- Clone balls

  Clone balls are the balls you can control in the game. You can change its moving direction at any time if you want. In addition, it can eat other balls smaller than itself by covering others' center. Once a clone ball eat other balls, its size will raise and it radius will increase accordingly. A clone ball has there skills:

  - Eject

    Ejecting a spore ball can help a clone ball decrease its size and make it move faster. When a clone ball ejects, the new spore ball must appear on the clone ball's moving direction with a high speed and quickly slow down.

  - Split

Spliting helps a clone ball to split itself in two pieces. The two pieces has the same size. Remember that all pieces will be merged in certain time. If you want to move faster, you can split and turn your balls into smaller size in order to get higher speed limit.

2.3.3 Rules of Game

There are a few rules to be aware of as following.

1. Player-balls have a decay on size to ensure that they will not grow too large. For example, we set `size_decay=0.00001` in our default setting, which means that a player-ball's size will drop 0.001% in a state frame. Normaly we will have 20 state frames in a second, that means a player-ball's size will drop 0.02% in a second. If your player-ball's size is too large, you must eat more others to remain your size.
2. If a player's all balls are eat, it will respawn in somewhere randomly and immediately.
3. The player's vision will depends on its balls' positions. We calculate the centroid of a player, and get the smallest external square of all balls. After that, we expand this square as the final vision. To guarantee each player's vision, we also provide them with the basic vision even if all their balls gather together. For example, if the balls of a player are dispersed enough, a larger vision will be applied for this player.
4. Each ball has its own speed limit based on its size. In order to ensure the balance of the game, we let larger balls move slow and smaller balls move fast.

2.3.4 High-level Operations

2.4 GoBigger Engine Design

2.4.1 Overview

This section mainly explains the engine design of GoBigger, including detailed motion logic for various balls. If readers want to develop a new environment based on GoBigger, or want to learn more about the development details of GoBigger, they can read this section carefully. For the convenience of explanation, we will introduce the logic of each ball, and the interaction with other balls. If you are not familiar with the basic ball units in GoBigger, it is recommended to check the previous section (What is GoBigger) first. Note that the following descriptions are based on the settings of `st_t4p3`.

2.4.2 What all balls have in common

1. In GoBigger, you can see that different balls have different sizes. We define that each ball has a certain `score`, and the corresponding `radius` can be calculated from the score.

```
import math

def score_to_radius(score):
    return math. sqrt(score / 100 * 0.042 + 0.15)
```

2. Food balls are immovable, but thorn balls, spore balls, and clone balls all contain speed attributes, which means they can all move.
3. If there is a relationship between two balls that can be eaten and eaten (such as between clone balls, between clone balls and all other balls, between thorn balls and spore balls), then when the centers of the two balls overlap, it will be determined whether phagocytosis can occur by the scores of the balls. We stipulate that phagocytosis can only occur when the score of the large ball exceeds `1.3` times that of the small ball.

4. The center of all balls will not exceed the map boundary during the movement.
5. Our default in-game FPS is `20`. That is, the game state is updated `20` times in a second. Also, by default each call to `env.step` will advance the environment two frames, the user-provided action will be done in the first frame, and then the second frame will be given an empty action.

2.4.3 Food Ball

1. Food balls are neutral resources in the game. In the `st_t4p3` configuration, the number of food balls on the map at the start of the game will be `800`, and the maximum number is `900`. Every `8` frames, `(maximum quantity - current quantity) * 0.01` food balls will be replenished, and the number of food balls will not exceed the upper limit.
2. The initial score for the food ball is `100`. That is, if a clone eats a food ball, the clone's score will increase by `100`.

2.4.4 Thorn Ball

1. Thorn balls are also neutral resource in the game. In the `st_t4p3` configuration, the number of thorn balls on the map at the beginning of the game will be `9`, and the maximum number is `12`. Every `120` frames, round up `(maximum quantity - current quantity) * 0.2` thorn balls will be added in the map, and the number of thorn balls will not exceed the upper limit.
2. The initial score of the ball of thorns will be randomly selected from the range of `10000` to `15000`.
3. When the player ejects the spore ball to the thorn ball, if the spore ball is covered by the center of the thorn ball, the thorn ball will eat the spore ball, and at the same time generate an initial velocity of `10` in the direction of the spore ball's movement, and the velocity decays uniformly to `0` over `20` frames.

2.4.5 Spore Ball

1. The size of the spore ball is fixed and the score is fixed at `1400`.
2. When the spore ball is spit out, the speed is fixed at `30`, and the speed is uniformly attenuated to `0` in the `20` frame.

2.4.6 Clone Ball

The size of the clone ball increases as it continues to eat the ball. The initial score for the clone ball is `1000`. A single player can have a maximum of `16` clone balls. The `eject` skill can only be used when the score of each clone ball exceeds `3200`, and the `split` skill can only be used when the score exceeds `3600`.

Speed

The speed of the clone ball is a combination of three parts of the vector: the player's action, the centripetal force caused by the presence of multiple balls, and the gradually decaying speed caused by splitting or eating the thorn ball. The acceleration caused by player actions and centripetal force is multiplied by the length of each frame as the speed change amount at each frame.

1. Player Action: As defined by the action space, the player can provide any point `(x,y)` within a unit circle to change the speed of the clone ball. Specifically, GoBigger will first normalize this point to the inside of the unit circle (or to the circle if it is outside, otherwise leave it alone), and then multiply it by the weight `30` to give the acceleration.

2. Centripetal force: When the player has multiple clone balls, a centripetal force will be generated inside the multiple balls, which points to the center of mass. In fact, the centripetal force will not be used directly, it will be divided by the radius as the true centripetal force, and multiplied by the weight `10` to be the acceleration.
3. Gradually attenuated speed after splitting or eating the thorn ball: The new clone ball split by eating the thorn ball will have a new initial speed after the split. The modulo length of this velocity is related to the radius after splitting, specifically `(260 - 20 * radius) / 7`, and the direction is away from the center of the circle. The velocity decays evenly to `0` over `14` frames. If the Clone Ball is obtained through the split skill, the formula for calculating the modulo length of this initial velocity is `(95 + 19 * radius) / 7`.
4. The speed brought by player operation and centripetal force will be limited by the upper speed limit. The upper speed limit is related to the radius of the ball, and adjusts the upper speed limit by using the larger of the player action and centripetal force as the `ratio`. The specific formula is `(2.35 + 5.66 / radius) * ratio`.

Eating a thorn ball

1. Each time the Clone Ball eats the Thorns Ball, it will split up to a maximum of `10` new Clone Balls. For example, if a player currently has only two clone balls, and one of the clone balls eats a thorn ball, he will split new `10` clone balls, so the player has a total of `` 12`` Clone Balls; if a player currently has `10` Clone Balls, and one of them eats a Thorny Ball, he will split a new `6` Clone Ball, so this At that time the player has a total of `16` clone balls.
2. The maximum score of the new ball split by the Clone Ball is `5000`. For example, the current score of the clone ball is `23000`, he eats a thorn ball with a score of `10000`, then his score will become `33000`. At the same time, this avatar will split into new `10` avatars. According to the even distribution, the score of each avatar is `3000`.
3. After eating the ball of thorns and splitting, the positions of the new balls are evenly distributed around, and there will always be new ball clones in the horizontal position to the right of the original ball.

Split

1. The new split ball will be in the direction specified by the player. If there is no specified direction, it will appear in the direction of movement of the original ball.
2. The split skill will divide the score of the original ball equally between the two split balls.
3. The split ball will also have the speed of the original one.
4. Whether after splitting or eating the thorn ball, the player's clone ball (including triggering the split and eating the thorn ball, as well as the newly added clone ball after these two operations) will enter a cooldown period with a length of `20`. Clone balls in the cooldown period cannot trigger the operation of merging with their own clone balls. Additionally, after the Clone Ball completes a merge, it will re-enter the cooldown phase.

Eject

1. The spore balls generated after the sporulation operation is performed by the clone ball will appear in the direction specified by the player. If there is no specified direction, it will appear in the movement direction of the clone ball.
2. Each time the clone ball spit out a spore ball, it is equivalent to dividing `1400` from its own score and assigning it to a new spore ball.

2.5 Real-time Interaction with game

GoBigger allow users to play game on their personal computer in real-time. Serveral modes are supported for users to explore this game.

Note: If your version of GoBigger is v0.1.x, please upgrade for a better experience. You can use `pip install --upgrade gobigger` to access the latest version of GoBigger.

2.5.1 Standard setting with partial view

If you want to play real-time game on your PC on your own, you can launch a game with the following code:

```
python -m gobigger.bin.play --mode st --vision-type partial
```

In this mode, using mouse to control the your balls to move, `Q` means eject spore on your moving direction, `W` means split your balls, and `E` means stop all your balls and gather them together.

2.5.2 Standard setting with full view

If you want to play real-time game on your PC with your friends, you can launch a game with the following code:

```
python -m gobigger.bin.play --mode st --vision-type full
```

2.5.3 Independent action setting with partial view

In the independent action game mode, each ball of the player receives an action individually. But because it is more difficult to control, we still only receive an action based on the mouse and keyboard in this mode, and assign this action to all the player's balls. The game can be started with the following code:

```
python -m gobigger.bin.play --mode sp --vision-type partial
```

2.6 Space

2.6.1 Standard game mode

Standard game mode refers to the same game as agar, where the player can only provide one action per frame, and then all player balls will perform that one action.

Action Space

Since each ball controlled by the player can only move, sporulate, split, and stop, the action space of GoBigger is relatively simple:

```
action = [x, y, action_type]
```

- **`x, y`: is a point in the unit circle. `(x, y)` represents the player's manipulation of the ball's acceleration.**
  - GoBigger will normalize `(x, y)` to ensure that its modulus does not exceed 1.
  - If the user does not provide acceleration changes, `(None, None)` can be provided to indicate no changes to the movement.
- **`action_type`: Int**
  - `0`: means only move, actually every time `env.step` is called.
  - `1`: Represents ejecting in the given direction. If the direction is not specified (ie `(None, None)`), it is executed in the direction of movement.
  - `2`: represents splitting in the given direction. If the direction is not specified (ie `(None, None)`), it is executed in the direction of movement.

We hope that players can use the various skills of the clone ball more flexibly, and hope that the movement direction will not limit the choice of skills. Therefore, we allow the player to specify different directions and directions of movement when using the skill. For example, when the player's clone ball is moving to the right, if you want to spore down, you only need to specify `action_type=1` and specify `(x, y)` at the same time, and the clone ball can go to the side. Continue to move right, while spitting down spores. A simple example is given below.

In addition, some useful actions can be achieved by the clever combination of `x`, `y` and `action_type`. For example, setting `x` and `y` to both `None` when sporulating spores can achieve cross sporulation. E.g:

Submit actions to the environment

In the case of multiple players, it is necessary to specify the correspondence between each action and the player. The submitted actions should be a dictionary, and each `key-value-pair` should be a player's `id` and the action he needs to do in this frame. Therefore, the action can be submitted as follows:

```
team_infos = env.get_team_infos()
actions = {}
for team_id, player_ids in team_infos.items():
    actions.update({player_id: [random.uniform(-1, 1), random.uniform(-1, 1), -1] for␣
↪player_id in player_ids})
obs = env.step(actions)
```

Observation Space

After each step of the game, the user can get the game state under the vision of the clone ball.

```
obs, reward, done, info = env.step()
global_state, player_states = obs
```

`global_state` contains some global information, as follows:

```
{
    'border': [map_width, map_height], # map size
    'total_frame': total_frame, # The total number of frames in the whole game
    'last_frame_count': last_frame_count, # The current number of frames that have passed
    'leaderboard': { team_name: team_size } # Current leaderboard information, including␣
↪the score of each team. The team's score is the sum of the players' scores in the team
}
```

`player_states` contains the information that each player can get, according to `player_id`, as follows:

```
{
    player_id: {
        'rectangle': [left_top_x, left_top_y, right_bottom_x, right_bottom_y], # The␣
↪position of the view box in the global view
        'overlap': {
            'food': [[position.x, position.y, radius, score], ...], # Food ball␣
↪information in the field of view, which are position xy, radius, score
            'thorns': [[position.x, position.y, radius, score, vel.x, vel.y], ...], #␣
↪The information of the thorns ball in the field of view, namely position xy, radius,␣
↪score, current speed xy
            'spore': [[position.x, position.y, radius, score, vel.x, vel.y, owner], ...],
↪ # The spore ball information in the field of view, which are the position xy, radius,␣
↪score, Current speed xy, from player's id
            'clone': [[[position.x, position.y, radius, score, vel.x, vel.y, direction.x,
↪ direction.y,
                        player_id, team_id], ...], # Player ball information in the␣
↪field of view, namely position xy, radius, score, current speed xy, current direction␣
↪xy, belonging player id, belonging team id
        },
        'team_name': team_name, # The current player's team id
        'score': player_score, # current player's score
        'can_eject': bool, # Whether the current player can perform the spore action
        'can_split': bool, # Whether the current player can perform the split action
    },
    ...
}
```

The `overlap` in `player_states` represents the structured information about the ball currently in the player's field of view. `overlap` is a simple dictionary, each key-value pair represents information about a ball in view. `overlap` contains structured information about food balls, thorn balls, spore balls, and clone balls. Specifically, for example, we found that the content of the `food` field is `[[3.0, 4.0, 2, 2], ...]` (only the first element in the list is shown here for simplicity) , then the meaning is that in the player's field of vision, there is a food ball with a radius of `2` at the coordinates `(3.0, 4.0)`, and the score of this food ball is `2`.

Note that the length of the information list for each type of ball is indeterminate. For example, if there are `20` food balls in the current frame view, the length of the list corresponding to the current `food` is `20`. In the next frame, if the food ball in the field of view becomes `25`, the corresponding list length will become `25`. Additionally, if only a portion of a ball is in the player's field of view, GoBigger will also give information about the center and radius of the ball in `overlap`.

Reward

After each `step` of the game, the user can get the game's default reward.

```
_, reward, _, _ = env.step()
```

The default reward in the game is very simple, it is the difference between the total score of the player's current frame and the total score of the previous frame. Users can design more complex rewards based on the player's obseravtion.

Other Information

GoBigger provides statistics and puts this information in `info`. After each `step` of the game, the user can get it.

```
_, _, _, info = env.step()
```

Specifically, `info` is a dictionary, and in the context of `st_t2p2`, the following information can be obtained:

2.6.2 Independent Action Game Mode

The independent action game mode means that the player needs to provide action to all of his player balls every frame. Each player ball of the player can perform actions independently.

Action Space

The minimum action unit is the same as the standard game mode, except that when doing `env.step(actions)`, the format of `actions` should be as follows:

```
actions = {
    player_id: {
        ball_id: [x, y, action_type],
        ...
    },
    ...
}
```

The `ball_id` here can be determined from the `obs` obtained in each frame. Each `ball_id` will uniquely correspond to one of the player's clone balls.

Observation Space

Most of it is the same as the standard game mode, the only difference is that the clone ball part will add `ball_id` information. This information can be used to tell the player where the `ball_id` can be taken from when providing `actions`.

`player_states` is as follows:

```
{
    player_id: {
        ...
        'overlap': {
            ...
```

(continues on next page)

(continued from previous page)

```
            'clone': [[[position.x, position.y, radius, score, vel.x, vel.y, direction.x,
↪ direction.y,
                        player_id, team_id, ball_id], ...], # The player's ball␣
↪information in the field of view, namely position xy, radius, current speed xy,␣
↪current direction xy, player id, team id, ball id
        },
        ...
    }
}
```

2.7 GoBigger environment

2.7.1 Overview

This section mainly explains the environment design of GoBigger, including the definition of a given environment, and how to customize the corresponding environment.

2.7.2 Defined environment

GoBigger defines some basic environments, which can be generated by the following code

```
from gobigger.envs import create_env
env = create_env('st_t4p3')
```

Among them, the first parameter received by `create_env()` is a string, which is the name of the environment we have defined. Similarly, in addition to `st_t4p3`, we also have `st_t2p2`, `st_t3p2` to choose from. The differences between each environment are described in detail below.

1. `st_t4p3`: Created a game scene with four teams of three players each. This is the standard game scene we define, and parameters including the refresh speed of various balls in it have been adjusted to a relatively robust level. Users can use this environment to solve some cooperative and confrontational problems in complex spatial scenarios.
2. `st_t2p2`: Considering that `st_t4p3` may be too large, we provide a smaller game scene. In this scenario, there are only two teams, and each team has two players. Also, the map size, the number of food orbs and thorn balls have been reduced accordingly. In addition, in order to reduce the player's early development time, we set the player's first birth score to 13000 in `st_t2p2` to allow players to fight quickly, while this value is 1000 in `st_t4p3` .
3. `st_t3p2`: This is a medium environment between `st_t2p2` and `st_t4p3`, with three teams of two players each. Like `st_t2p2`, a higher birth score is also set to reduce development time.

In addition, the above environment defaults to an action every two frames (20 frames per second). If you want a longer action interval, you can set it with the following code:

```
from gobigger.envs import create_env
env = create_env('st_t4p3', step_mul=10)
```

In addition, we have more environments already defined, as follows:

Name	Agents Size	Map Size	Food	Thorn	Init Size	Limited Frame
Small Maps						
st_t1p1	1x1	32x32	[65,75]	[1,2]	1000	3600(3min)
st_t1p2	1x2	48x48	[130,150]	[2,3]	1000	3600(3min)
st_t2p1	2x1	48x48	[130,150]	[2,3]	1000	3600(3min)
st_t2p2	2x2	64x64	[260,300]	[3,4]	13000	3600(3min)
st_t3p2	3x2	88x88	[500,560]	[5,6]	13000	3600(3min)
Large Maps						
st_t4p3	4x3	128x128	[800,900]	[9,12]	1000	14400(12min)
st_t5p3	5x3	128x128	[900,1000]	[10,12]	1000	14400(12min)
st_t5p4	5x4	144x144	[900,1000]	[10,12]	1000	14400(12min)
st_t6p4	6x4	144x144	[1000,1100]	[11,13]	1000	14400(12min)

2.7.3 Customize the corresponding environment

GoBigger's rich configuration file design allows users to easily set up every detail in the game.

If you want to make a simple modification on the basis of an environment we have defined, for example, modify the duration of a round in `st_t2p2`, you can do the following:

```
from gobigger.envs import create_env
env = create_env('st_t2p2', dict(
    frame_limit=10*60*20,
))
```

In this way, the new environment opened will become a 10-minute round based on `st_t2p2`.

2.8 Playback System

GoBigger's playback system supports three options, which can be selected through the environment's configuration file. The configuration items involved are as follows:

```
config = dict(
    ...
    playback_settings=dict(
        playback_type='none', # ['none', 'by_video', 'by_frame']
        by_video=dict(
            save_video=False,
            save_fps=10,
            save_resolution=552,
            save_all=True,
            save_partial=False,
            save_dir='.',
            save_name_prefix='test',
        ),
        by_frame=dict(
            save_frame=False,
            save_all=True,
            save_partial=False,
            save_dir='.',
```

(continues on next page)

(continued from previous page)

```
            save_name_prefix='test',
        ),
        by_action=dict(
            save_action=False,
            save_dir='.',
            save_name_prefix='test',
        ),
    ),
    ...
)
```

playback_type can be one of ['none', 'by_video', 'by_frame'].

- none: means no need to save playback
- by_video: means to save the video directly, and the suffix of the saved file is .mp4. Generally speaking, the video saved in the st_t4p3 environment is about 80M.
- by_frame: Represents the change amount of each frame saved, and the suffix of the saved file is .pb. Generally speaking, the st_t4p3 environment saves files around 25M.

2.8.1 Save video

If playback_type='by_video' is selected, the specific configuration items can be as follows:

```
env = create_env('st_t4p3', dict(
    playback_settings=dict(
        playback_type='by_video',
        by_video=dict(
            save_video=True,
            save_dir='.', # The directory location where the video needs to be saved
            save_name_prefix='test', # Save the prefix of the video name
        ),
    ),
))
```

2.8.2 Save the pb file

If playback_type='by_frame' is selected, the specific configuration items can be as follows:

```
env = create_env('st_t4p3', dict(
    playback_settings=dict(
        playback_type='by_frame',
        by_frame=dict(
            save_frame=True,
            save_dir='.', # The directory location where the video needs to be saved
            save_name_prefix='test', # The prefix of the name of the saving video
        )
    ),
))
```

After getting the saved .pb file, you need to view it through our given player. Execute the following command in the command line to open the player.

```
python -m gobigger.bin.replayer
```

After opening the player, you need to select the `.pb` file you want to view. Then you can start watching. The player supports double-speed playback, including 2x, 4x, and 8x (by clicking the button in the lower left corner). Also supports dragging the progress bar.

2.9 Configuration

2.9.1 Overview

In order to allow players to further understand the realization of the game mechanism, we have opened some parameters for players to adjust. We hope that players can modify the corresponding parameters to achieve a variety of different environments. At the same time, you can also design your own agent environment to quickly verify the algorithm.

GoBigger puts configurable parameters in `gobigger/server/server_default_config.py`

2.10 Collision Detection Algorithm

2.10.1 Overview

In order to detect the collision of the sphere in each frame of the game so that the server can update the state of the sphere, we need to design an efficient collision detection algorithm.
We have designed four collision detection algorithms and encapsulated them into the following four classes.

2.10.2 Algorithm efficiency analysis

The theoretical time complexity of the above four algorithms is as follows:

ExhaustiveCollisionDetection:

$O(n*m)$

where n denotes the total number of balls and m denotes the number of balls to be asked.

PrecisionCollisionDetection:

$O(n*log(n)+\Sigma r*logn+p)$

where n denotes the total number of balls, m denotes the number of balls to be asked, k denotes the presicion we set and p denotes number of spheres that actually collided.

RebuildQuadTreeCollisionDetection:

$O(n*log(n)+m*log(n)+p)$

where n denotes the total number of balls, m denotes the number of balls to be asked and p denotes number of spheres that actually collided.

RemoveQuadTreeCollisionDetection:

$O(r*log(n)+m*log(n)+p)$

where n denotes the total number of balls, m denotes the number of balls to be asked, r denotes the number of spheres whose position status has changed and p denotes number of spheres that actually collided.

In order to test the efficiency of these algorithms, we modify the parameters including the total number of balls, the number of queries, the number of changed balls, and the iteration rounds to set the test scenario. The data in the following table comes from the most representative scenarios

- *T: the number of all balls in the map*
- *Q: the number of query balls, which usually means the moving balls in the map*
- *C: the number of changing balls, which means the number of collisions*

	Exhaustive	Precision	Rebuild QuadTree	Remove QuadTree

T=3000
Q=300
C=600
	688ms	14ms	47ms	48ms

T=3000
Q=300
C=1500
	1067ms	16ms	50ms	178ms

T=10000
Q=1000
C=2000
	8384ms	61ms	339ms	497ms

T=10000
Q=2000
C=5000
	12426ms	86ms	586ms	2460ms

T=30000
Q=6000
C=3000
	127000ms	403ms	5691ms	8419ms

In order to see the pros and cons of each algorithm more intuitively, we integrated test data and drew diagrams of four algorithms and various parameters as follows:

According to the results, we can think that the **PrecisionCollisionDetection** algorithm is far better than the other algorithms in terms of efficiency and stability.

2.11 FAQ

2.11.1 Q1: How to save video in a game?

A1

Set `playback_settings` when you create the environment. After finishing a game, `test.pb` will be saved at `save_dir`, which can be shown by the GoBigger replayer.

```
env = create_env('st_t3p2', dict(
    playback_settings=dict(
        playback_type='by_frame',
        by_frame=dict(
            save_frame=True,
            save_dir='.',
            save_name_prefix='test',
        ),
    ),
))
```

2.11.2 Q2: What are the winning conditions at the end of the game?

A2

Sort by calculating the sum of the scores of all players under each team at the end of the game.

2.11.3 Q3: Is there a limit to the size of the partial field of view?

A3

The size of the player's partial field of view is determined by the relative position of the player's doppelganger. We set the player's minimum field of view to be a matrix of 36*36. With the dispersion of the doppelganger, the player's maximum field of vision can reach the global level.

2.12 agents

2.12.1 BaseAgent

`class gobigger.agents.base_agent.BaseAgent`

Overview:
The base class of all agents

2.12.2 BotAgent

`class gobigger.agents.bot_agent.BotAgent`(*name=None*, *level=3*)

Overview:
A simple script bot

2.13 balls

2.13.1 BaseBall

`class gobigger.balls.base_ball.BaseBall`(*ball_id*, *position*, *score*, *border*, ***kwargs*)

Overview:
Base class of all balls

`check_border()`

Overview:
Check to see if the position of the ball exceeds the bounds of the map. If it exceeds, the speed and acceleration in the corresponding direction will be zeroed, and the position will be edged

`static default_config()`

Overview:
Default config

`eat`(*ball*)

Overview:
Describe the rules of eating and being eaten

Parameters:
ball <BaseBall>: Eaten ball

`get_dis`(*ball*)

Overview:
Get the distance between the centers of the two balls

Parameters:
ball <BaseBall>: another ball

`judge_cover`(*ball*)

Overview:
Determine whether the center of the two balls is covered

Parameters:
ball <BaseBall>: another ball

Returns:
is_covered <bool>: covered or not

`judge_in_rectangle`(*rectangle*)

Overview:
Determine if the ball and rectangle intersect

Parameters:
rectangle <List>: left_top_x, left_top_y, right_bottom_x, right_bottom_y

Returns:
<bool> : intersect or not

`move`(*direction*, *duration*)

Overview:
Realize the movement of the ball, pass in the direction and time parameters, and return the new position

Parameters:
direction <Vector2>: A point in the unit circle duration <float>: time

Returns:
position <Vector2>: position after moving

`remove`()

Overview:
Things to do when being removed from the map

2.13.2 FoodBall

class `gobigger.balls.food_ball.`**`FoodBall`**(*ball_id*, *position*, *score*, *border*, ***kwargs*)

Overview:

- characteristic:

- Can’t move, can only be eaten, randomly generated

static **`default_config`**()

Overview:
Default config

`eat`(*ball*)

Overview:
Describe the rules of eating and being eaten

Parameters:
ball <BaseBall>: Eaten ball

`move`(*direction*, *duration*)

Overview:
Realize the movement of the ball, pass in the direction and time parameters, and return the new position

Parameters:
direction <Vector2>: A point in the unit circle duration <float>: time

Returns:
position <Vector2>: position after moving

2.13.3 ThornsBall

class `gobigger.balls.thorns_ball.`**`ThornsBall`**(*ball_id*, *position*, *score*, *border*, ***kwargs*)

Overview:

- characteristic:

- Can’t move actively
- Can eat spores. When eating spores, it will inherit the momentum of the spores and move a certain distance.
- Can only be eaten by balls heavier than him. After eating, it will split the host into multiple smaller units.
- Nothing happens when a ball lighter than him passes by

static **`default_config`**()

Overview:
Default config

`eat`(*ball*)

Overview:
Describe the rules of eating and being eaten

Parameters:
ball <BaseBall>: Eaten ball

`move`(*direction=None*, *duration=0.05*, ***kwargs*)

Overview:
Realize the movement of the ball, pass in the direction and time parameters, and return the new position

Parameters:
direction <Vector2>: A point in the unit circle duration <float>: time

Returns:
position <Vector2>: position after moving

2.13.4 CloneBall

`class gobigger.balls.clone_ball.CloneBall`(*ball_id*, *position*, *score*, *border*, *team_id*, *player_id*, *vel_given=<Vector2(0, 0)>*, *acc_given=<Vector2(0, 0)>*, *from_split=False*, *from_thorns=False*, *split_direction=<Vector2(0, 0)>*, *spore_settings={'score_init': 1.5, 'vel_init': 50, 'vel_zero_frame': 10}*, *sequence_generator=None*, ***kwargs*)

Overview:
One of the balls that a single player can control - characteristic: * Can move * Can eat any other ball smaller than itself * Under the control of the player, the movement can be stopped immediately and contracted towards the center of mass of the player * Skill 1: Split each unit into two equally * Skill 2: Spit spores forward * There is a percentage of weight attenuation, and the radius will shrink as the weight attenuates

`static default_config`()

Overview:
Default config

`eat`(*ball*, *clone_num=None*)

Parameters:
clone_num <int>: The total number of balls for the current player

`eject`(*direction=None*) → list

Overview:
When spit out spores, the spores spit out must be in the moving direction of the ball, and the position is tangent to the original ball after spitting out

Returns:
Return a list containing the spores spit out

`judge_rigid`(*ball*)

Overview:
Determine whether two balls will collide with a rigid body

Parameters:
ball <CloneBall>: another ball

Returns:
<bool>: collide or not

`move`(*given_acc=None*, *given_acc_center=None*, *duration=0.05*)

Overview:
Realize the movement of the ball, pass in the direction and time parameters

`on_thorns`(*split_num*) → list

Overview:
Split after encountering thorns, calculate the score, position, speed, acceleration of each ball after splitting

Parameters:
split_num <int>: Number of splits added

Returns:
Return a list that contains the newly added balls after the split, the distribution of the split balls is a circle and the center of the circle has a ball

`rigid_collision`(*ball*)

Overview:
When two balls collide, We need to determine whether the two balls belong to the same player A. If not, do nothing until one party is eaten at the end B. If the two balls are the same owner, judge whether the age of the two is full or not meet the fusion condition, if they are satisfied, do nothing. C. If the two balls are the same owner, judge whether the age of the two is full or not meet the fusion condition, Then the two balls will collide with rigid bodies This function completes the C part: the rigid body collision part, the logic is as follows:

1. To determine the degree of fusion of the two balls, use [the radius of both] and subtract [the distance between the two] as the magnitude of the force
2. Calculate the coefficient according to the weight, the larger the weight, the smaller the coefficient will be
3. Correct the position of the two according to the coefficient and force

Parameters:
ball <CloneBall>: another ball

Returns:
state <bool>: the operation is successful or not

`score_decay`()

Overview:
Control the score of the ball to decay over time

`split`(*clone_num*, *direction=None*) → list

Overview:
Active splitting, the two balls produced by splitting have the same volume, and their positions are tangent to the forward direction

Parameters:
clone_num <int>: The total number of balls for the current player

Returns:
The return value is the new ball after the split

2.13.5 SporeBall

class gobigger.balls.spore_ball.**SporeBall**(*ball_id*, *position*, *border*, *score*, *direction=<Vector2(0, 0)>*, *owner=-1*, ***kwargs*)

Overview:
Spores spit out by the player ball - characteristic: * Can't move actively * can not eat * Can be eaten by CloneBall and ThornsBall * There is an initial velocity at birth, and it decays to 0 within a period of time

static default_config()

Overview:
Default config

eat(*ball*)

Overview:
Describe the rules of eating and being eaten

Parameters:
ball <BaseBall>: Eaten ball

move(*direction=None*, *duration=0.05*)

Overview:
Realize the movement of the ball, pass in the direction and time parameters, and return the new position

Parameters:
direction <Vector2>: A point in the unit circle duration <float>: time

Returns:
position <Vector2>: position after moving

2.14 managers

2.14.1 BaseManager

class gobigger.managers.base_manager.**BaseManager**(*cfg*, *border*)

Overview:
Base class for all ball managers

add_balls(*balls*)

Overview:
Add one (or more) balls

get_balls()

Overview:
Get all balls currently managed

obs()

Overview:
Return data available for observation

`refresh()`

Overview:
Refresh. Used to refresh the balls in management. Such as replenishing eaten food balls

`remove_balls`(*balls*)

Overview:
Remove managed balls

`step`(*duration*)

Overview:
Perform a status update under the control of the server

2.14.2 FoodManager

class `gobigger.managers.food_manager.FoodManager`(*cfg*, *border*, *random_generator=None*, *sequence_generator=None*)

`add_balls`(*balls*)

Overview:
Add one (or more) balls

`get_balls()`

Overview:
Get all balls currently managed

`refresh()`

Overview:
Refresh. Used to refresh the balls in management. Such as replenishing eaten food balls

`remove_balls`(*balls*)

Overview:
Remove managed balls

`step`(*duration*)

Overview:
Perform a status update under the control of the server

2.14.3 ThornsManager

class `gobigger.managers.thorns_manager.ThornsManager`(*cfg*, *border*, *random_generator=None*, *sequence_generator=None*)

`add_balls`(*balls*)

Overview:
Add one (or more) balls

`get_balls()`

Overview:
Get all balls currently managed

`refresh()`

Overview:
Refresh. Used to refresh the balls in management. Such as replenishing eaten food balls

`remove_balls(`*balls*`)`

Overview:
Remove managed balls

`step(`*duration*`)`

Overview:
Perform a status update under the control of the server

2.14.4 PlayerManager

`class gobigger.managers.player_manager.PlayerManager(`*cfg*, *border*, *team_num*, *player_num_per_team*, *spore_manager_settings*, *random_generator=None*, *sequence_generator=None*`)`

`add_balls(`*balls*`)`

Overview:
Add one (or more) balls

`adjust()`

Overview:
Adjust all balls in all players

`get_balls()`

Overview:
Get all balls currently managed

`get_player_names()`

Overview:
get all names of players

`get_player_names_with_team()`

Overview:
get all names of players by teams

`get_team_names()`

Overview:
get all names of players by teams with team names

`remove_balls`(*balls*)

Overview:
Remove managed balls

`reset`()

Overview:
reset manager

`step`()

Overview:
Perform a status update under the control of the server

2.14.5 SporeManager

class gobigger.managers.spore_manager.**SporeManager**(*cfg*, *border*, *random_generator=None*, *sequence_generator=None*)

`add_balls`(*balls*)

Overview:
Add one (or more) balls

`get_balls`()

Overview:
Get all balls currently managed

`remove_balls`(*balls*)

Overview:
Remove managed balls

`step`(*duration*)

Overview:
Perform a status update under the control of the server

2.15 players

2.15.1 BasePlayer

class gobigger.players.base_player.**BasePlayer**(*name=None*)

Player's abstract class

`eat`(*ball*)

Eat another ball

`eject`()

Do sporulation

`move(direction)`

Parameters:
direction <Vector2>: Given any point in a unit circle, the angle represents the direction, and the magnitude represents the acceleration

`stop()`

stop moving

2.15.2 HumanPlayer

class `gobigger.players.human_player.HumanPlayer`(*cfg*, *team_id*, *player_id*, *border*, *spore_settings*, *sequence_generator=None*)

`add_balls(balls)`

Overview:
Add new avatars

Parameters:
balls <List[CloneBall] or CloneBall>: It can be a list or a single doppelganger

`adjust()`

Overview:
Adjust all the balls controlled by the player, including two parts 1. Possible Rigid Body Collision 2. Possible ball-ball fusion

`cal_centroid()`

Overview:
Calculate the centroid

`eat(ball)`

Eat another ball

`eject(direction=None)`

Overview:
All clones controlled by the player perform the spore-spitting action

Return:
<list>: list of new spores

`get_balls()`

Overview:
Get all the balls of the current player

`get_clone_num()`

Overview:
Get how many avatars the current player has

`get_keys_sort_by_balls()`

Overview:
Sort by ball score from largest to smallest

Return:
<list>: list of names

`get_total_score()`

Overview:
Get the total score of all balls of the current player

`move(`*direction=None*, *duration=0.05*`)`

Overview:
Move all balls controlled by the player The main logic is

1. Processing stopped state
2. If it is stopping, control all balls to move closer to the center of mass

Parameters:
direction <Vector2>: A point in the unit circle duration <float>: time

Returns:
position <Vector2>: position after moving

`score_decay()`

Overview:
The player's balls' scor will decay over time

`split(`*direction=None*`)`

Overview:
All avatars controlled by the player perform splits, from large to small

2.16 render

2.16.1 BaseRender

`class gobigger.render.base_render.BaseRender(`*game_screen_width*, *game_screen_height*, *info_width=0*, *info_height=0*, *with_show=False*`)`

2.16.2 EnvRender

`class gobigger.render.env_render.EnvRender(`*game_screen_width=512*, *game_screen_height=512*, *info_width=60*, *info_height=0*, *with_show=False*, *padding=20*, *map_width=256*, *map_height=256*`)`

Overview:
No need to use a new window, giving a global view and the view that each player can see

2.16.3 RealtimeRender

class gobigger.render.realtime_render.**RealtimeRender**(*game_screen_width=512*, *game_screen_height=512*, *info_width=0*, *info_height=0*, *with_show=True*, *padding=20*, *map_width=128*, *map_height=128*)

Overview:
Used in real-time games, giving a global view

2.16.4 RealtimePartialRender

class gobigger.render.realtime_render.**RealtimePartialRender**(*game_screen_width=512*, *game_screen_height=512*, *info_width=0*, *info_height=0*, *with_show=True*)

Overview:
Used in real-time games to give the player a visible field of view. The corresponding player can be obtained by specifying the player name. The default is the first player

2.17 server

2.17.1 Server

class gobigger.server.server.**Server**(*cfg=None*, *seed=None*)

spawn_balls()

Overview:
Initialize all balls. If self.custom_init is set, initialize all balls based on it.

2.18 utils

2.18.1 Border

class gobigger.utils.structures.**Border**(*minx*, *miny*, *maxx*, *maxy*, *random_generator=None*)

Overview:
used to specify a rectangular range

contains(*position: Vector2*) → bool

Overview:
To judge whether a position in this border.

Parameters:
position <Vector2>: the position to be judged.

Returns:
bool: True or False, whether the position in this border.

`sample()` → Vector2

Overview:
Randomly sample a position in the border.

Returns:
Vector2: the sampled position.

2.18.2 BaseCollisionDetection

class `gobigger.utils.collision_detection.`**`BaseCollisionDetection`**(*border:* Border)

2.18.3 ExhaustiveCollisionDetection

class `gobigger.utils.collision_detection.`**`ExhaustiveCollisionDetection`**(*border:* Border)

Overview:
Exhaustive Algorithm

`solve`(*query_list: list*, *gallery_list: list*)

Overview:
For the balls in the query, enumerate each ball in the gallery to determine whether there is a collision

Parameters:
query_list <List[BaseBall]>: List of balls that need to be queried for collision gallery_list <List[BaseBall]>: List of all balls

Returns:

results <Dict[int: List[BaseBall]> return value

int value denotes:
the subscript in query_list

string value denotes:
List of balls that collided with the query corresponding to the subscript

2.18.4 PrecisionCollisionDetection

class `gobigger.utils.collision_detection.`**`PrecisionCollisionDetection`**(*border:* Border, *precision: int = 50*)

Overview:
Precision Approximation Algorithm Divide the map into several rows according to the accuracy that has been set, dynamically maintain the row information in each frame, and search by row

`get_row`(*x*) → int

Overview:
Get the row coordinates of the ball

Parameter:
node <BaseBall>: The ball need to get its row coordinates

`solve`(*query_list: list*, *gallery_list: list*)

Overview:
First, you need to sort the balls in each row according to the ordinate. For the balls in query_list, first abstract the boundary of the ball into a rectangle, then traverse each row in the rectangle, and find the first ball covered by the query through dichotomy in each row, and then Enumerate the balls in sequence until the ordinate exceeds the boundary of the query rectangle.

Parameters:
query_list <List[BaseBall]>: List of balls that need to be queried for collision gallery_list <List[BaseBall]>: List of all balls

Returns:

results <Dict[int: List[BaseBall]> return value

int value denotes:
the subscript in query_list

string value denotes:
List of balls that collided with the query corresponding to the subscript

2.18.5 RebuildQuadTreeCollisionDetection

`class gobigger.utils.collision_detection.RebuildQuadTreeCollisionDetection`(*border:* Border, *node_capacity=64*, *tree_depth=32*)

Overview:
Build a quadtree on a two-dimensional plane in every frame, and query collisions in the quadtree

`solve`(*query_list: list*, *gallery_list: list*)

Overview:
Construct a quadtree from scratch based on gallery_list and complete the query

Parameters:
query_list <List[BaseBall]>: List of balls that need to be queried for collision gallery_list <List[BaseBall]>: List of all balls

Returns:

results <Dict[int: List[BaseBall]> return value

int value denotes:
the subscript in query_list

string value denotes:
List of balls that collided with the query corresponding to the subscript

2.18.6 RemoveQuadTreeCollisionDetection

`class gobigger.utils.collision_detection.RemoveQuadTreeCollisionDetection`(*border:* Border, *node_capacity=64*, *tree_depth=32*)

Overview:
Add delete operations for the quadtree, and dynamically maintain a quadtree

`solve`(*query_list: list*, *changed_node_list: list*)

Overview:
Update the points in the quadtree according to the changed_node_list and complete the query

Parameters:
query_list <List[BaseBall]>: List of balls that need to be queried for collision gallery_list <List[BaseBall]>: List of all balls

Returns:

results <Dict[int: List[BaseBall]> return value

int value denotes:
the subscript in query_list

string value denotes:
List of balls that collided with the query corresponding to the subscript

INDEX