Chapter 17
Arena Platform for Multi-Agent Reinforcement Learning

Zihan Ding

Abstract In this chapter, we introduce a project named *Arena* for multi-agent reinforcement learning research. The hands-on instructions are provided in this chapter for building games with *Arena* toolkit, including a single agent game and a simple two-agent game with different reward schemes. The reward scheme in *Arena* is a way to specify the social structure among multiple agents, which contains social relationships of non-learnable, isolated, competitive, collaborative, and mixed types. Different reward schemes can be applied at the same time in a hierarchical structure in one game scene, together with the individual-to-group structure for physical units, to describe the complex relationships in multi-agent systems comprehensively. Moreover, we also show the process of applying the baseline in *Arena*, which provides several implemented multi-agent reinforcement learning algorithms as a benchmark. Through this project, we want to provide the readers with a useful tool for investigating multi-agent intelligence with customized game environments and multi-agent reinforcement learning algorithms.

Keywords Multi-agent reinforcement learning · Learning environment · Toolkit · Competitive · Collaborative · Social relationship

In this chapter, we will introduce a powerful toolkit for multi-agent reinforcement learning (MARL): *Arena* (Song et al. 2019). *Arena* is a general evaluation platform for multi-agent intelligence based on Unity, with learning environments of diverse logic and representations, as well as easy configurations on complex social tree relationships between multiple agents. *Arena* also contains the implementation of state-of-the-art deep multi-agent reinforcement learning algorithm baselines, which can help the users to quickly testify the built-up environments. Generally, *Arena* is a building toolkit for researchers to easily invent and build unexplored multi-agent problems with customized game environments. The official website of *Arena*

Z. Ding (✉)
Imperial College London, London, UK
e-mail: zhding@mail.ustc.edu.cn

H. Dong et al. (eds.), *Deep Reinforcement Learning*,
https://doi.org/10.1007/978-981-15-4095-0_17

is: https://sites.google.com/view/arena-unity/home. *Arena* focuses on first-person or third-person action games, leveraging the fantastic rendering effects of Unity. Recently published open-sourced project OpenSpiel (https://github.com/deepmind/open_spiel) by DeepMind, focuses on multi-agent board or card games.

There are two major modules in *Arena*: (1). the Building Toolkit, which is used to quickly build multi-agent environments with customized characteristics; (2). the Baselines, for testing the built-up environments with MARL algorithms. We will start by building the environments in *Arena*.

17.1 Intallation

Unity ML-agents toolkit is a prerequisite for *Arena*, which needs to be installed before applying *Arena*. The complete process of installing *Arena* follows the official website of Building Toolkit[1] and Baseline.[2]

Note that if you are running on a remote server without a graphical user interface (e.g., X-Server) or you cannot get access to the X-Server, you will need to set up the virtual display following instructions in Sect. 17.3.1 or guidelines on the official website of *Arena*.

After installation, we can find that in the `Arena-BuildingToolkit/Assets/ArenaSDK/GameSet/` file of *Arena* folder, there are dozens of built-in games with both continuous and discrete action spaces. They are pre-designed as examples for using *Arena*, you can read the scripts of all those games for better understanding of how *Arena* works. All games and abstraction layers share one Unity project. Each game is held in an independent folder, with the game's name as the folder name. The folder `ArenaSDK` holds all the abstraction layers and shared code, assets, and utilities. Code style is kept as consistent as possible to the Unity ML-agents toolkit.

17.2 Build Game with *Arena*

We will go through making a multi-agent game with the *Arena* Building Toolkit. It will not require much coding work with many off-the-shelf assets and multi-agent features managed by *Arena*. Before you start, we are expecting you to have some basic knowledge about Unity. Therefore, you are recommended to finish the roll-a-ball tutorial (https://learn.unity.com/project/roll-a-ball-tutorial) to learn all the basic concepts of Unity.

[1]Building Toolkit: https://github.com/YuhangSong/Arena-BuildingToolkit.

[2]Baseline: https://github.com/YuhangSong/Arena-Baselines.

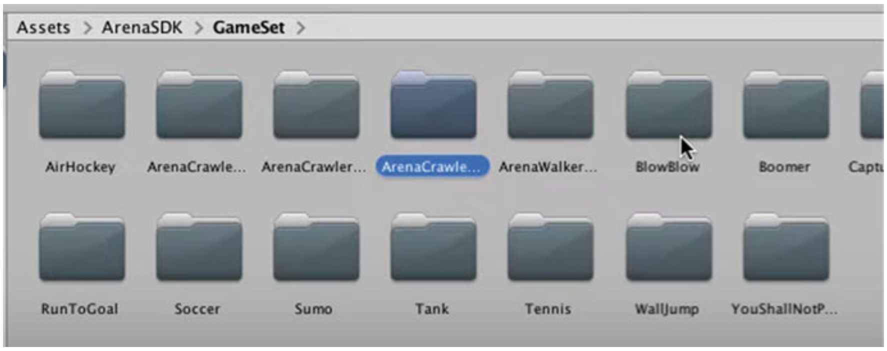


Fig. 17.1 Built-in games in *Arena* file

In order to use *Arena*, run Unity, choose open project, and select the cloned or downloaded “Arena-BuildingToolkit” file. The opening process may take some time for the first time.

We can see dozens of built-in games in the *Arena* folder, as shown in Fig. 17.1. They are pre-designed as examples for using *Arena*, you can read the scripts of all those games for better understanding of how *Arena* works. We will provide basic instructions on building these games in the following sections.

17.2.1 Simple One-Player Game

We start by building a basic game environment with only one player in it:

- Create a folder to host your game. In this part, we create a folder named “1P” for only one-player game.
- On the left-side “Hierarchy” window, we delete the original **Main Camera** and **Directional Light** in it. Drag the prefab **GlobalManager** built in *Arena* folder `Assets/ArenaSDK/SharedPrefabs` as shown in Fig. 17.2, to the left-side “Hierarchy” window, as shown in Fig. 17.3. Note that the prefabs are useful and shared components in Unity for using any built-in objects with a simple dragging operation. **GlobalManager** in *Arena* manages the whole game, so all the other components need to be attached under it.
- Next we need to place a playground for the agent to play on, we find the prefabs in *Arena* called **PlayGroundWithDeadWalls**, and attach it to the child called **World** of **GlobalManager**. The **GlobalManager** also has another child **TopDownCamera** for providing an overall view of the game. This step is shown in Fig. 17.4.
- Similar as above, we need to attach a **BasicAgent** from *Arena* prefabs and attach it to the **GlobalManager** as shown in Fig. 17.5. So now we have one agent on the playground in the scene, and we can manually drag the agent to a proper position,

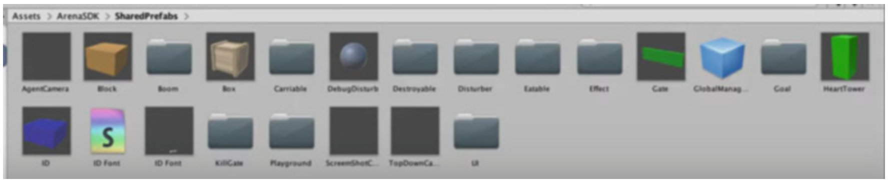


Fig. 17.2 The *Arena* built-in prefabs

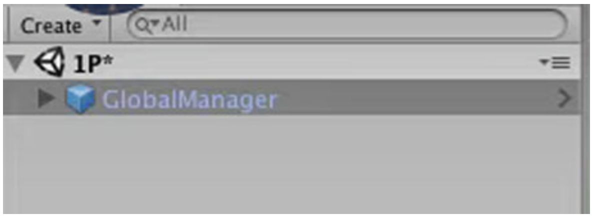


Fig. 17.3 Drag the **GlobalManager** in *Arena* prefabs to the "Hierarchy" window of current game

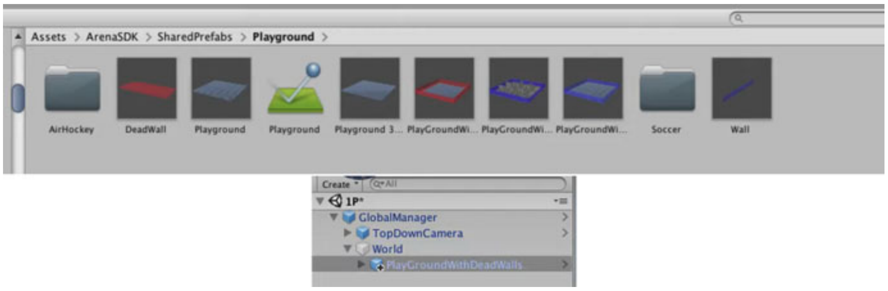


Fig. 17.4 Choose a playground in *Arena* prefabs and attach it to the child of the **GlobalManager**

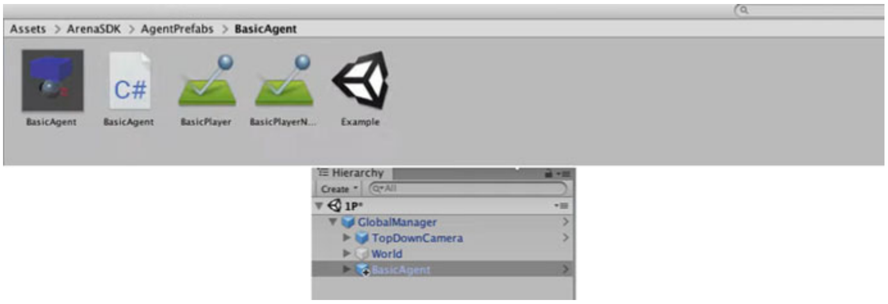


Fig. 17.5 Choose and attach a **BasicAgent** in *Arena* prefabs and attach it to the child of the **GlobalManager**

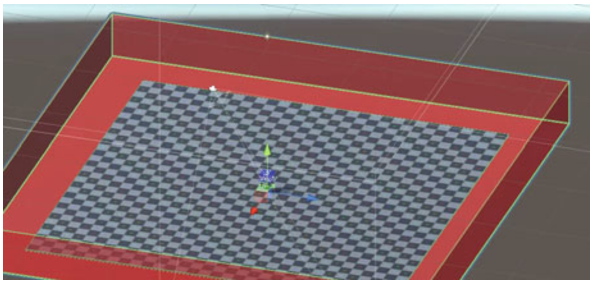
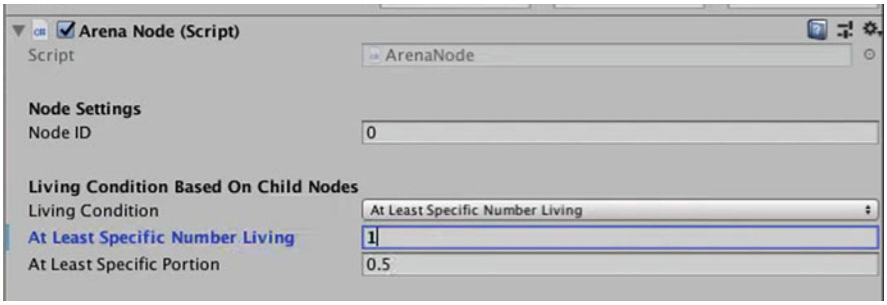


Fig. 17.6 The scene with a single agent on the playground



Arena Node (Script)

Script	ArenaNode
Node Settings	
Node ID	0
Living Condition Based On Child Nodes	
Living Condition	At Least Specific Number Living
At Least Specific Number Living	1
At Least Specific Portion	0.5

Fig. 17.7 Configure the game settings of a single-player game

as in Fig. 17.6. The values of the x, y, and z coordinates of position and rotation will be displayed on the **Transform** property of the agent.

- In order to make the game work normally, we also need to configure the game parameters as shown in Fig. 17.7. Here we only need to change the **Living Condition Based On Child Nodes** of the **GlobalManager**. The **Living Condition** is chosen to be **At Least Specific Number Living** and the **At Least Specific Number Living** value is set to be 1. As we only have one agent in this game, the above settings ensure that whenever the number of agents under **GlobalManager** is smaller than one, the game episode will end and restart. Now we can press the **Play** button to play the game and operate the agent with keys "W, A, S, D." As on the edges of the playground are the "dead walls," whenever the agent touches it, it will die and the game will restart. There are lots of other properties if you apply the **BasicAgent**, including different **Actions Settings**, **Reward Functions** (for reinforcement learning), etc. You can play around with them (only the **Actions Settings** are valid in this simple game) to get familiar with *Arena* Building Toolkit.

17.2.2 *Simple Two-Player Game with Reward Scheme*

In this section, we will introduce how to deploy more than one agent in the game environment with a social tree.

- First, let us start from the above single-player game. If we choose the **Global-Manager** or the **BasicAgent**, we will find that for both objects there is a script called **Arena Node (Script)** as shown in Figs. 17.8 and 17.9, which is a basic concept used to define the social relationships in *Arena* games. Descriptions about **Arena Node** will be provided in this section.
- We choose **BasicAgent** built before and duplicate it by pressing Ctrl+C and Ctrl+V in the left hierarchy window, as shown in Fig. 17.10. Now we have two **Arena Node**s under the **Global Manager**, therefore we need to set the **Node ID** to be 1 instead of 0 for any one of the **BasicAgent**s in order to discriminate them (Fig. 17.11). The positions of the agents in the scene can be moved to a proper position, so as to separate them because the two agents are initialized at the same position after duplication.

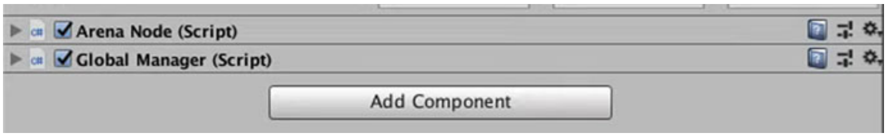


Fig. 17.8 The **Arena Node (Script)** exists in **GlobalManager**

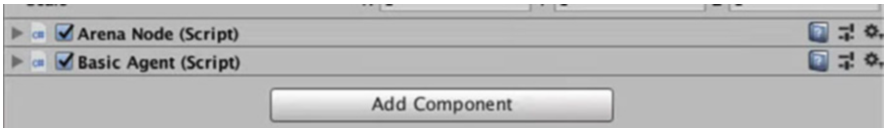


Fig. 17.9 The **Arena Node (Script)** exists in **BasicAgent**

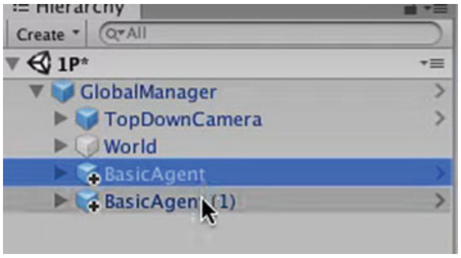
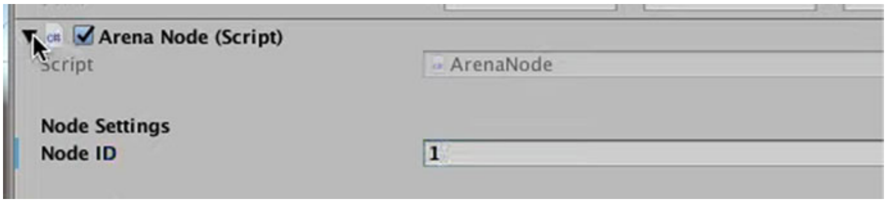
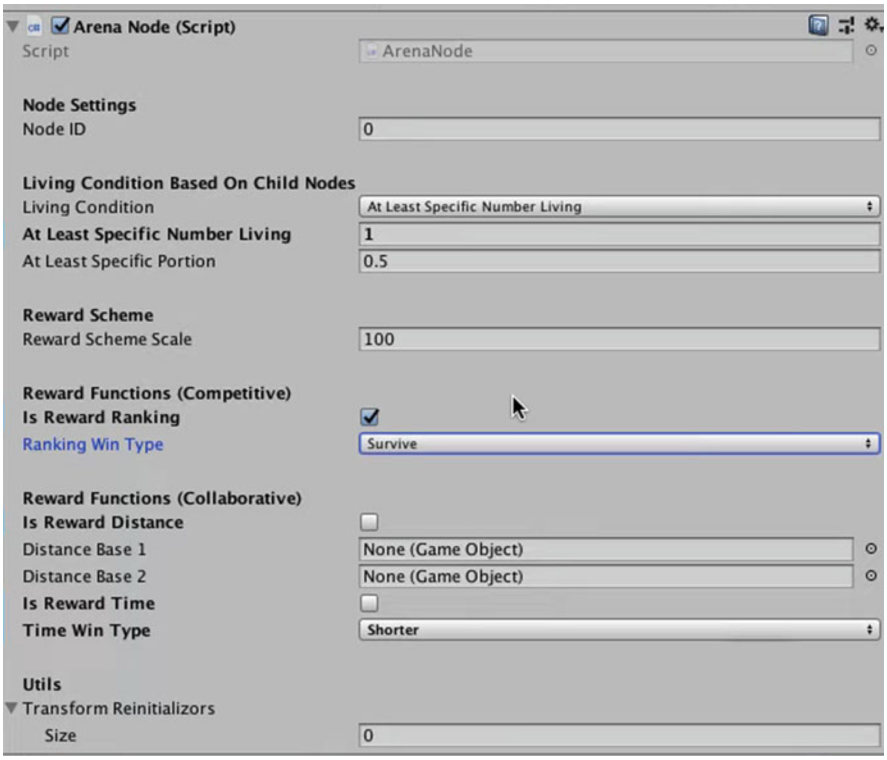


Fig. 17.10 Duplicate the **BasicAgent** under the **GlobalManager**


Arena Node (Script)
Script: ArenaNode

Node Settings
Node ID: 1

Fig. 17.11 Change node ID to be different from each other when there are multiple nodes under **GlobalManager**


Arena Node (Script)
Script: ArenaNode

Node Settings
Node ID: 0

Living Condition Based On Child Nodes
Living Condition: At Least Specific Number Living
At Least Specific Number Living: 1
At Least Specific Portion: 0.5

Reward Scheme
Reward Scheme Scale: 100

Reward Functions (Competitive)
Is Reward Ranking: ✓
Ranking Win Type: Survive

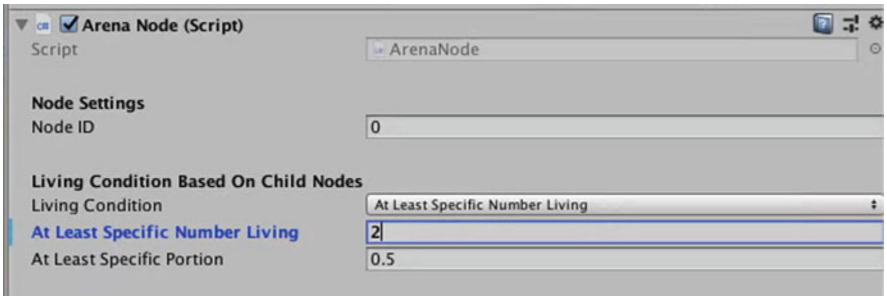
Reward Functions (Collaborative)
Is Reward Distance: ☐
Distance Base 1: None (Game Object)
Distance Base 2: None (Game Object)
Is Reward Time: ☐
Time Win Type: Shorter

Utils
Transform Reinitializors
Size: 0

Fig. 17.12 Set the reward scheme under **GlobalManager**

- Next we choose **GlobalManager, Arena Node (Script)**, and we can set the reward functions for the game, as shown in Fig. 17.12. We click the **Is Reward Ranking**, which is a competitive reward function for agents under **GlobalManager**. We also choose **Ranking Win Type** to be **Survive**, which means the agent died at the last place (surviving in the end) will get a positive reward. If you select **Depart**, the reward will be given to the agent died in the first place. We also need to unclick **Is Reward Distance** (which gives dense reward values according to the distance from the agent to the target) and **Is Reward Time** (which gives reward values according to the living time of the agent). Above are different reward schemes built in *Arena*, in a competitive and/or collaborative manner. Different games will have different reward settings to represent different social structures. You can play around with different reward settings for different games. For example, if you want to set a dense reward according to the distance between the agent and the target for a task like reaching the target, you need to click **Is Reward Distance**, as well as dragging a target object to the **Target** blank to make it work.

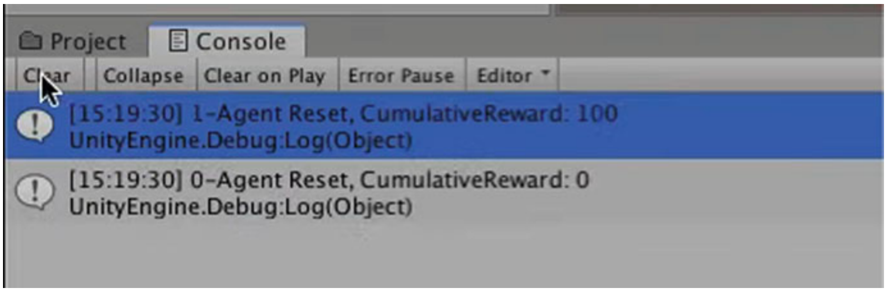
- We need to set the **At Least Specific Number Living** to be 2 under the **Living Condition Based On Child Nodes** of **GlobalManager**, as shown in Fig. 17.13. So that only when at least two agents are living, the game will continue; otherwise the game will end and restart. Now we click the **Play** button, the game should work normally. As long as one agent died, the game will end and rewards/penalties will be given to agents as displayed in the **Console**, as shown in Fig. 17.14.
- Next we will make the game more complex, we want to have two teams with each containing two agents to compete with each other. So, first, we create an empty object at the hierarchy window and name it “2 Player Team”. Then we attach the **Arena Node** script to it, as shown in Fig. 17.15.


Arena Node (Script)
Script: ArenaNode

Node Settings
Node ID: 0

Living Condition Based On Child Nodes
Living Condition: At Least Specific Number Living
At Least Specific Number Living: 2
At Least Specific Portion: 0.5

Fig. 17.13 Set the least number of living agents in **GlobalManager**



```
[15:19:30] 1-Agent Reset, CumulativeReward: 100
UnityEngine.Debug:Log(Object)
[15:19:30] 0-Agent Reset, CumulativeReward: 0
UnityEngine.Debug:Log(Object)
```

Fig. 17.14 The rewards given to each agent are displayed at the **Console**

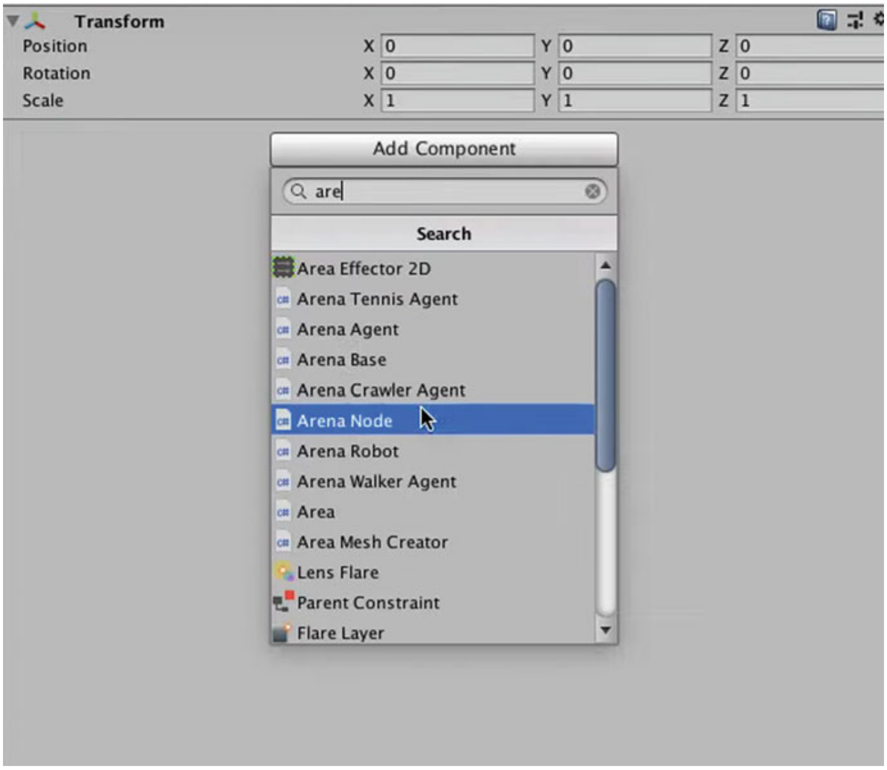


Fig. 17.15 Attach the **Arena Node** script to the team object

- Now we drag the two previous **BasicAgent** to the new-built team object **2 Player Team**. Then we duplicate the **2 Player Team**, change the **Node ID** of the second team object to be 1 instead of 0. Now we have a structure of teams and agents as shown in Fig. 17.16. If we click the **Play** button now, we shall see two teams with two agents each in the scene as Fig. 17.17.
- As the **At Least Specific Number Living** of **GlobalManager** is set to be 2, any team's death will cause the game to end. Also as the **At Least Specific Number Living** of **2 Player Team** is 1 as default, only when both two agents died at the same team will cause the team to die. We can also set the game logic to be different, if we set the **Living Condition** of the **2 Player Team** to be **All Living**, then any agent died in a team will cause the team to die, and therefore end the game. From above, you can see that with the social tree structure like **GlobalManager->Team->Agent**, *Arena* can basically support any kinds of social relationships with the **Arena Node** through defining the living and reward schemes.

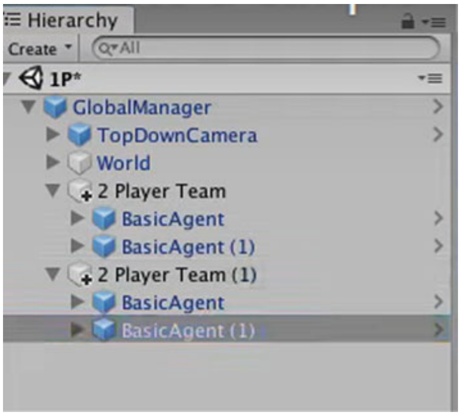


Fig. 17.16 The hierarchy structure of two teams with two agents each under the **GlobalManager** in *Arena*

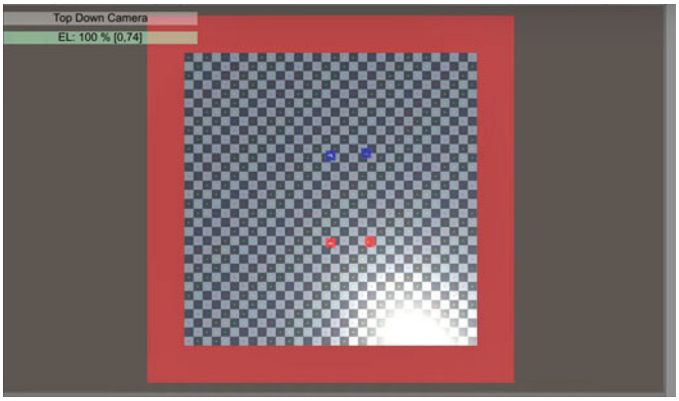


Fig. 17.17 The game scene of two teams with two agents each during the play

17.2.3 Advanced Settings

Reward Scheme

To construct complex social tree relationship, there are five basic multi-agent reward schemes (BMaRSs) in *Arena* that define different social paradigms at each node in the social tree, including: **non-learnable (NL), isolated (IS), competitive (CP), collaborative (CL), competitive and collaborative mixed** (CC). Specifically, each BMaRS is a restriction applied to the reward function, so it corresponds to a batch of reward functions that can lead to a specific social paradigm. For each BMaRS, *Arena* provides multiple ready-to-use reward functions (sparse and dense), simplifying the construction of games with complex social relationships. Apart from providing reward functions, *Arena* also offers a verification option for customized reward functions, so that one can make sure that the programmed reward functions lie in one of the BMaRSs and produce a specific social paradigm. We will introduce these five different reward schemes in detail.

First we need to give some preliminaries. We consider a Markov game as defined in the basics of RL, consisting of multiple agents $x \in \mathcal{X}$, a finite global state space $s_t \in \mathcal{S}$, a finite action space $a_{x,t} \in \mathcal{A}_x$ for each agent x, and a bounded-step reward space $r_{x,t} \in \mathbb{R}$ for each agent x. For the environment, it consists of a transition function $g : \mathcal{S} \times \{\mathcal{A}_x : x \in \mathcal{X}\} \to \mathcal{S}$, which is a stochastic (due to the stochasticity of Unity simulator) function $s_{t+1} \sim g\big(s_{t+1} | (s_t, \{a_{x,t} : x \in \mathcal{X})\}\big)$, a reward function for each agent $f_x : \mathcal{S} \times \{\mathcal{A}_x : x \in \mathcal{X}\} \to \mathbb{R}$, which is a deterministic function $r_{x,t+1} = f_x\big(s_t, \{a_{x,t} : x \in \mathcal{X}\}\big)$, a joint reward function $f = \{f_x : x \in \mathcal{X}\}$, and episode reward $R_x^f = \sum_{t=1}^{T} r_{x,t}$ for each agent x under the joint reward function f. For the agent, *Arena* considers that it observes $s_{x,t} \in \mathcal{S}_x$, where $\mathcal{S}_x$ consists of a part of the information from the global state space $\mathcal{S}$. Therefore, there is a policy $\pi_x : \mathcal{S}_x \to \mathcal{A}_x$, which is a stochastic function $a_{x,t} \sim \pi_x(s_{x,t})$. Besides, *Arena* considers that agent x can take a policy π_x from a set of policies Π_x. *Arena* assumes that the random seed of all sampling operations is k, which is sampled from the whole seed space $\mathcal{K}$.

The definitions of different BMaRSs employ the basic concepts including agents $\{x : x \in \mathcal{X}\}$, policies $\{\pi_x : \Pi_x\}$, agent rewards $\{R_x^f : x \in \mathcal{X}\}$, and joint reward functions $\mathcal{F} = \{f : \cdot\}$ on population $\mathcal{X}$. The five different BMaRSs in *Arena* are defined in the following way:

1. **Non-learnable (NL)** BMaRSs ($\mathcal{F}^{NL}$) are a set of joint reward functions f as follows:

$$\mathcal{F}^{NL} = \big\{f : \forall k \in \mathcal{K}, \forall x \in \mathcal{X}, \forall \pi_x \in \Pi_x, \partial R_x^f / \partial \pi_x = \mathbf{0}\big\}, \tag{17.1}$$

   where $\mathbf{0}$ is a zero matrix of the same size and shape as the parameter space that defines π_x. Intuitively, $\mathcal{F}^{NL}$ means R_x^f for any agent $x \in \mathcal{X}$ is not optimizable by improving its policy π_x.

2. **Isolated (IS)** BMaRSs ($\mathcal{F}^{IS}$) are a set of joint reward functions f as follows:

$$\mathcal{F}^{IS} = \Bigg\{f : f \notin \mathcal{F}^{NL} \text{ and } \forall k \in \mathcal{K}, \forall x \in \mathcal{X}, \forall x' \in \mathcal{X} \setminus \{x\}, \forall \pi_x \in \Pi_x, \forall \pi_{x'} \in \Pi_{x'}, \frac{\partial R_x^f}{\partial \pi_{x'}} = \mathbf{0}\Bigg\}, \tag{17.2}$$

   where "\" is the set difference. Intuitively, $\mathcal{F}^{IS}$ means that the episode reward R_x^f received by any agent $x \in \mathcal{X}$ is not related to any policy $\pi_{x'}$ taken by any other agent $x' \in \mathcal{X} \setminus \{x\}$. Reward functions f_x in f of $\mathcal{F}^{IS}$ are often called *internal reward functions* in other multi-agent approaches (Hendtlass 2004; Jaderberg et al. 2018; Bansal et al. 2018), meaning that apart from the reward functions applied at a population level (such as win/lost), which are too sparse to learn, there are also reward functions directing the learning process towards receiving the population-level rewards, yet are more frequently available, i.e., more dense (Singh et al. 2009, 2010; Heess et al. 2017). $\mathcal{F}^{IS}$ is especially practical when the agent is a robot requiring continuous control of applying force on each of its joints, which means basic motor skills (such as moving) need to be learned before generating population-level intelligence. Therefore, *Arena* provides f in $\mathcal{F}^{IS}$ of: energy cost, punishment of applying a big force, encouragement of keeping a steady velocity, and moving distance towards the target.

3. **Competitive (CP)** BMaRSs ($\mathcal{F}^{CP}$) are inspired by Cai and Daskalakis (2011) and defined as

$$\mathcal{F}^{CP} = \Bigg\{ f : f \notin \mathcal{F}^{NL} \cup \mathcal{F}^{IS} \text{ and } \forall k \in \mathcal{K}, \forall x \in \mathcal{X}, \\ \forall \pi_x \in \Pi_x, \forall \pi_{x'} \in \Pi_{x'}, \frac{\partial \int_{x' \in \mathcal{X}} R^f_{x'} dx'}{\partial \pi_x} = \mathbf{0} \Bigg\}, \tag{17.3}$$

which intuitively means that for any agent $x \in \mathcal{X}$, taking any possible policy $\pi_x \in \Pi_x$, the sum of the episode reward of all agents will not change. If the episode length is 1, it expresses a classic multi-player zero-sum game (Cai and Daskalakis 2011).

Useful examples of f within $\mathcal{F}^{CP}$ are: (1) agents fight for a limited amount of resources that are always exhausted at the end of the episode, and the agent is rewarded for the amount of resources that it gained; and (2) fight till death, and the reward is given based on the order of death (the reward can also be based on the reversed order, so that the one departing the game first receives the highest reward, such as in some poker games, the one who first discards all cards wins). *Rock, Paper, and Scissors* in normal-form game (Myerson 2013) and *Cyclic Game* in Balduzzi et al. (2019) are both special cases of $\mathcal{F}^{CP}$;

4. **Collaborative (CL)** BMaRSs ($\mathcal{F}^{CL}$) are inspired by Cai and Daskalakis (2011) and defined as

$$\mathcal{F}^{CL} = \Bigg\{ f : f \notin \mathcal{F}^{NL} \cup \mathcal{F}^{IS} \text{ and } \forall k \in \mathcal{K}, \forall x \in \mathcal{X}, \\ \forall x' \in \mathcal{X} \setminus \{x\}, \forall \pi_x \in \Pi_x, \forall \pi_{x'} \in \Pi_{x'}, \frac{\partial R^f_{x'}}{\partial R^f_x} \geq 0 \Bigg\}, \tag{17.4}$$

which intuitively means that there is no conflict of interest ($\partial R^f_{x'} / \partial R^f_x < 0$) for any pair of agents (x', x). Besides, since $f \notin \mathcal{F}^{NL} \cup \mathcal{F}^{IS}$, there is at least one pair of agents (x, x') that makes $\partial R^f_{x'} / \partial R^f_x > 0$. This indicates that this pair of agents shares a common interest, so that improving R^f_x for agent x means improving $R_{x'}^{f}$ for agent x'. The most common example of f within $\mathcal{F}^{CL}$ is that f_x for all $x \in \mathcal{X}$ is identical, such as the moving distance of an object that can be pushed forward by the joint effort of multiple agents, or the alive duration of the population (as long as there is at least one agent alive in the population, the population is alive). Therefore, *Arena* provides f in $\mathcal{F}^{CL}$: living time of the team (both positive and negative, since some games require the team to survive as long as possible, while other games require the team to depart as early as possible, such as poker).

5. **Competitive and Collaborative Mixed (CC)** BMaRSs ($\mathcal{F}^{CC}$) are defined to be any situation other than the above four ones.

$$\mathcal{F}^{CC} = \left\{f : f \notin \mathcal{F}^{NL} \cup \mathcal{F}^{IS} \cup \mathcal{F}^{CP} \cup \mathcal{F}^{CL}\right\}. \tag{17.5}$$

First, the term $\partial \int_{x' \in \mathcal{X}} R_{x'}^{f} dx' / \partial \pi_x = \mathbf{0}$ in (17.3) can be written as $\int_{x' \in \mathcal{X}} \partial R_{x'}^{f} / \partial R_x^{f} dx' = 0$ (proof is not provided here, which refers to original paper), which makes an alternative (17.3). Considering $\mathcal{F}^{CP}$ in this alternative (17.3) and $\mathcal{F}^{CL}$ in (17.5), an intuitive explanation of $\mathcal{F}^{CC}$ is that there exist circumstances when $\partial R_{x'}^{f} / \partial R_x^{f} < 0$, meaning that the agents are competitive at this point. But the derivative of total interest $\int_{x' \in \mathcal{X}} \partial R_{x'}^{f} / \partial R_x^{f} dx'$ is not always 0, therefore, the total interest can be maximized with specific policies, meaning that the agents are collaborative at this point.

Apart from providing several practical f in each BMaRS, *Arena* also provides a verification option for each BMaRS, meaning that one can customize a f and use this verification option to make sure that the programmed f lies in a specific BMaRS.

The above contents provide the theory about how to use different kinds of reward functions to define social relationships. Moreover, the reward functions should be defined with respect to the categories of the above definitions to achieve the expected social relationships in the population. In practice, the reward functions have some specific formats like those we mentioned in previous sections. The *Arena* framework usually defines the **Collaborative** and **Competitive** reward functions in the **Arena Node** of **GlobalManager**, and the **Isolated** reward functions are defined in the **Arena Node** of agents like **BasicAgent**.

Here is an example for understanding the social tree relationship using different BMaRs for each **Arena Node** as shown in Fig. 17.18.[3] The reward schemes are assigned at each **Arena Node** to define the social relationships of its sub-nodes. The graphical user interface (GUI) in Fig. 17.18a defines a tree structure in Fig. 17.18b, representing a population of 4 agents. The tree structure can be easily reconfigured through dragging, duplicating, or deleting in the GUI in Fig. 17.18a. In this example,

[3]Figure source: Song, Yuhang, et al. "Arena: A General Evaluation Platform and Building Toolkit for Multi-Agent Intelligence." arXiv preprint arXiv:1905.08085 (2019).

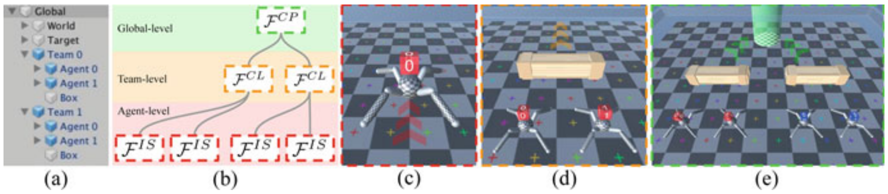


Fig. 17.18 The social tree defined in *Arena* using different BMaRs for each **Arena Node**

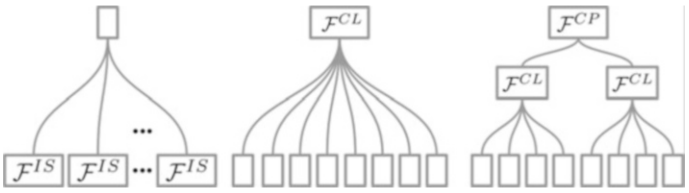


Fig. 17.19 Common social paradigms defined in *Arena* framework

each agent has an agent-level BMaRS. The agent is a robot ant, so that the agent-level BMaRSs are $\mathcal{F}^{IS}$, specifically, the option of *ant-motion* that directs the learning towards basic motion skills such as moving forward, as shown in Fig. 17.18c. Each two agents form a *team* (which is a set of agents or teams), and the two agents have team-level BMaRSs. In this example, the two robot ants collaborate with each other to push a box forward, as shown in Fig. 17.18d. Therefore, the team-level BMaRSs are $\mathcal{F}^{CL}$, specifically, the moving distance of the box. On the two teams, *Arena* has global-level BMaRSs. In this example, the two teams are set to have a match regarding which team pushes its box to the target point first, as shown in Fig. 17.18e. Therefore, the global-level BMaRSs are $\mathcal{F}^{CP}$, specifically, the ranking of the box reaching the target. The final reward function applied to each agent is a weighted sum of the above three BMaRSs at three levels. One can imagine defining a social tree of more than three levels, where small teams form into bigger teams, and BMaRSs are defined at each node to give more complex and structured social problems. After defining the social tree and applying BMaRSs on each node, the environment is ready for use with the abstraction layer handling everything else, such as assigning viewports to each agent in the window, applying the team color, displaying the agent ID, and generating a top-down view.

Moreover, we can easily extend the above framework to other common social paradigms, as shown in Fig. 17.19.[4]

[4]Figure source: Song, Yuhang, et al. "Arena: A General Evaluation Platform and Building Toolkit for Multi-Agent Intelligence." arXiv preprint arXiv:1905.08085 (2019).

More Agent Prefabs

Apart from the previous **BasicAgent**, *Arena* has other more advanced agent prefabs for the off-the-shelf usage, as shown in Fig. 17.20. The usage of other agents is mostly like **BasicAgent**, through dragging and attaching it under the **GlobalManager**. The only difference lies in the action space, you need to change a corresponding brain for controlling different agents. For example, the **ArenaCrawlerAgent** in the agent prefabs looks like Fig. 17.21, which has continuous action space for controlling the action values of joints. In order to use this agent properly, we need to change the brain of **ArenaCrawlerAgent** to be **ArenaCrawlerPlayerContinuous (PlayerBrain)** as shown in Fig. 17.22. Then the game can be exported and used for training as general games.

17.2.4 Export Binary Game

After you have tested the game in Unity within the player mode, and make sure that there is no problem in the game settings, you can export the game to be a standalone binary file, and use it for training the MARL algorithms with Python scripts. This section will show you how to export the game.

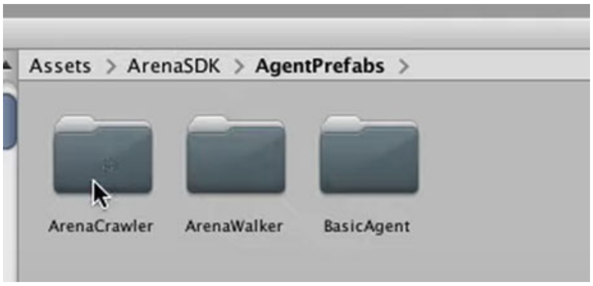


Fig. 17.20 Different agent prefabs in *Arena*

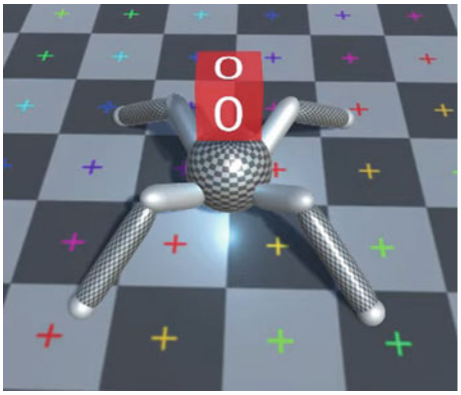


Fig. 17.21 The **ArenaCrawlerAgent** in the scene

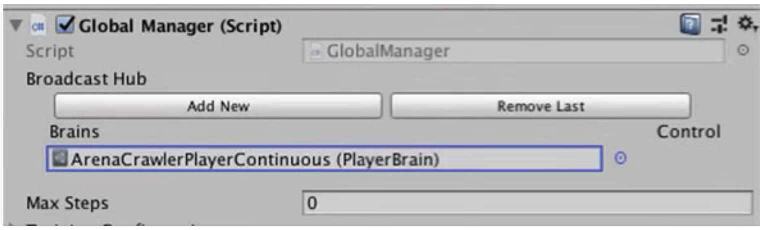

Global Manager (Script)
Script GlobalManager
Broadcast Hub
Add New Remove Last
Brains Control
ArenaCrawlerPlayerContinuous (PlayerBrain)
Max Steps 0

Fig. 17.22 Change the brain for **ArenaCrawlerAgent**

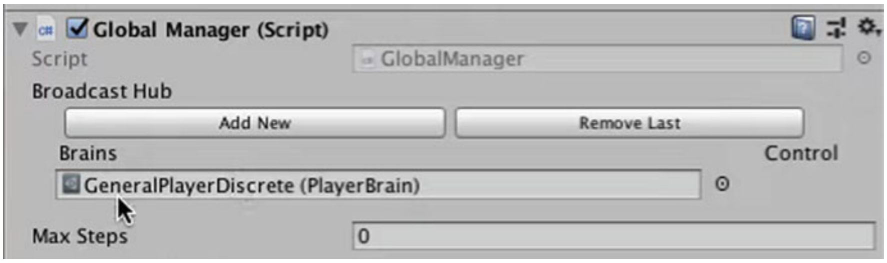
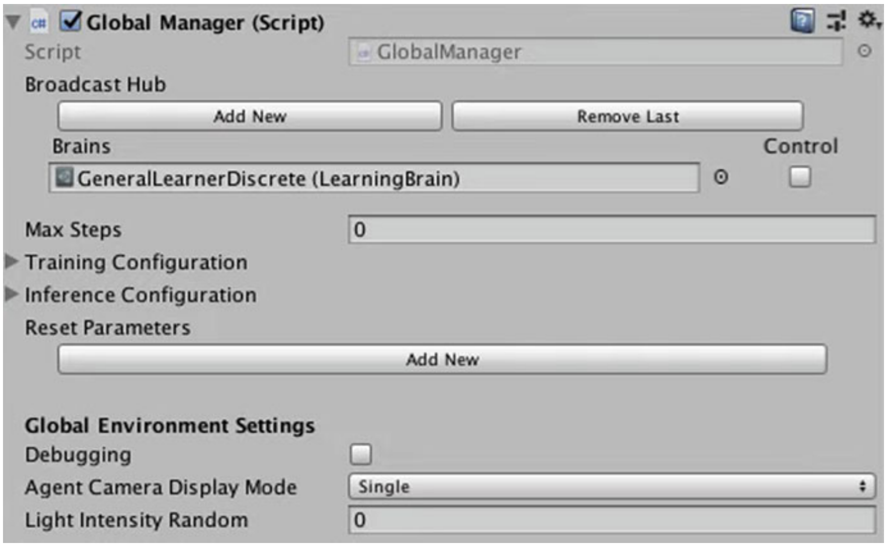

Global Manager (Script)
Script GlobalManager
Broadcast Hub
Add New Remove Last
Brains Control
GeneralPlayerDiscrete (PlayerBrain)
Max Steps 0

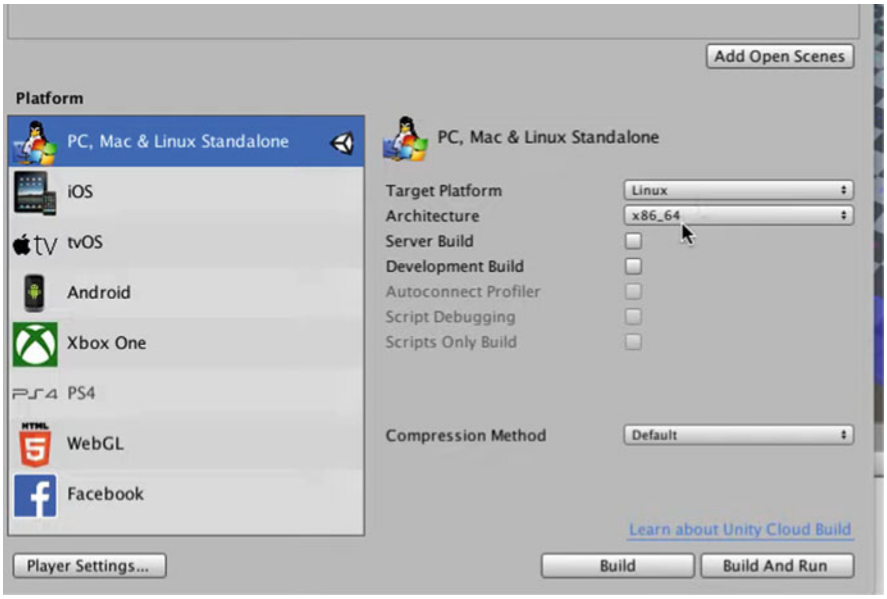
Fig. 17.23 The original brain type in player mode

- First, we need to change the brain type from **PlayerBrain** to a corresponding **LearningBrain** (of the same type), the **PlayerBrain** is used for controlling the game agents with user keyboard operations and the **LearningBrain** is to learn the controlling with learning algorithms. As shown in Fig. 17.23, for this game, we change the **GeneralPlayerDiscrete (PlayerBrain)** to be the **GeneralLearnerDiscrete (LearningBrain)** in Fig. 17.24. We also uncheck the **Debugging** to reduce output information during training.
- To export the game, we choose File->Build Settings, and get a window like in Fig. 17.25. We set the **Target Platform** and **Architecture** accordingly.
- We also need to click **Player Settings** to check the configurations, as shown in Fig. 17.26. One thing to notice is that the **Display Resolution Dialog** needs to be **Disabled** to work correctly. Then we go back to the previous window and click **Build**. We can get the binary file of the game after building.



Global Manager (Script)
Script: GlobalManager
Broadcast Hub
Add New | Remove Last
Brains: GeneralLearnerDiscrete (LearningBrain) — Control: ☐
Max Steps: 0
Training Configuration
Inference Configuration
Reset Parameters
Add New
Global Environment Settings
Debugging: ☐
Agent Camera Display Mode: Single
Light Intensity Random: 0

Fig. 17.24 Change the brain type to be **LearningBrain** to export the game for training



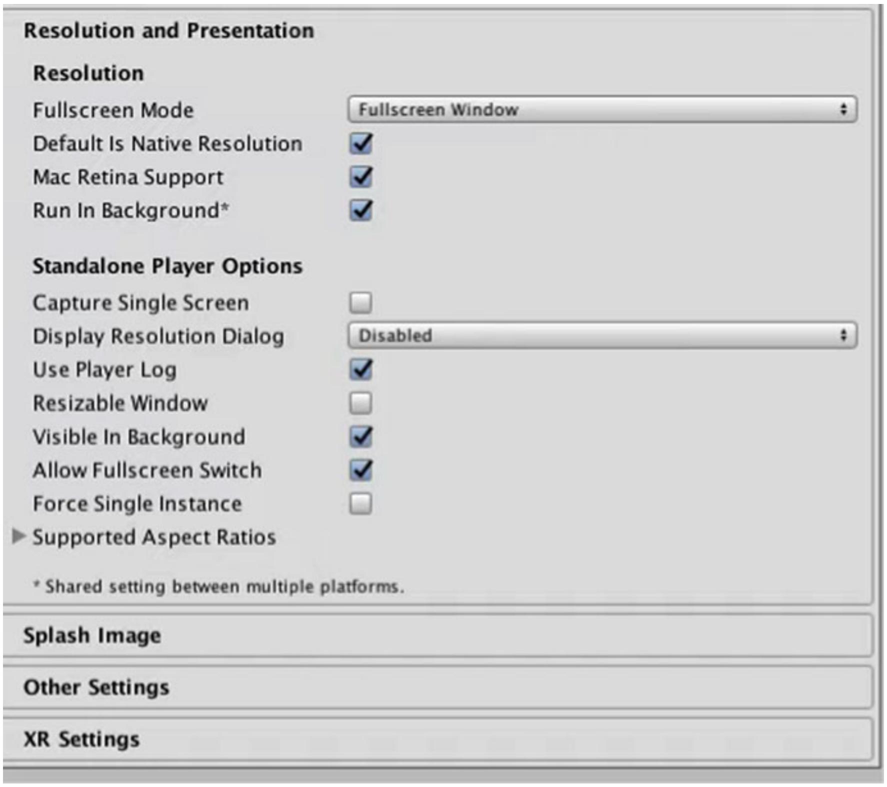
Add Open Scenes

Platform
- PC, Mac & Linux Standalone
- iOS
- tvOS
- Android
- Xbox One
- PS4
- WebGL
- Facebook

PC, Mac & Linux Standalone
Target Platform: Linux
Architecture: x86_64
Server Build: ☐
Development Build: ☐
Autoconnect Profiler: ☐
Script Debugging: ☐
Scripts Only Build: ☐
Compression Method: Default

Learn about Unity Cloud Build
Player Settings... | Build | Build And Run

Fig. 17.25 Check the configurations when building the game



Resolution and Presentation

Resolution

Fullscreen Mode	Fullscreen Window
Default Is Native Resolution	✓
Mac Retina Support	✓
Run In Background*	✓

Standalone Player Options

Capture Single Screen	☐
Display Resolution Dialog	Disabled
Use Player Log	✓
Resizable Window	☐
Visible In Background	✓
Allow Fullscreen Switch	✓
Force Single Instance	☐

▶ Supported Aspect Ratios

* Shared setting between multiple platforms.

Splash Image

Other Settings

XR Settings

Fig. 17.26 The window of configuring export of the game

17.3 MARL Training

With the exported stand-alone games built with *Arena*, we can set up the training process for investigating diverse problems in multi-agent reinforcement learning (MARL).

Before training, we need to first set up the system. As MARL generally requires large amounts of computation, we usually need a server to handle the training process. The basic settings of *Arena* environments follow the Sect. 17.1 at the beginning of this chapter. If you cannot use the X-Server properly on the server, you can follow the following subsection to setup virtual display; otherwise, just jump to the training sections.

17.3.1 Setup X-Server

The basic settings of using virtual display are as follows:

```
# Install Xorg
sudo apt-get update
sudo apt-get install -y xserver-xorg mesa-utils
sudo nvidia-xconfig -a --use-display-device=None
    --virtual=1280x1024

# Get the BusID information
nvidia-xconfig --query-gpu-info

# Add the BusID information to your /etc/X11/xorg.conf file
sudo sed -i 's/ BoardName "GeForce GTX TITAN X"/ BoardName
    "GeForce GTX TITAN X"\n BusID "0:30:0"/g' /etc/X11/xorg.conf

# Remove the Section "Files" from the /etc/X11/xorg.conf file
# And remove two lines that contain Section "Files" and
    EndSection
sudo vim /etc/X11/xorg.conf

# Download and install the latest Nvidia driver for ubuntu
# Please refer to
    http://download.nvidia.com/XFree86/Linux-#x86_64/latest.txt
wget http://download.nvidia.com/XFree86/Linux-x86_64/390.87/
    NVIDIA-Linux-x86_64-390.87.run
sudo /bin/bash ./NVIDIA-Linux-x86_64-390.87.run --accept-license
    --no-questions --ui=none

# Disable Nouveau as it will clash with the Nvidia driver
sudo echo 'blacklist nouveau' | sudo tee -a
    /etc/modprobe.d/blacklist.conf
sudo echo 'options nouveau modeset=0' | sudo tee -a
    /etc/modprobe.d/blacklist.conf
sudo echo options nouveau modeset=0 | sudo tee -a
    /etc/modprobe.d/nouveau-kms.conf
sudo update-initramfs -u

sudo reboot now
```

Kill Xorg using one of the following three ways (different ways may work on different Linux versions):

```
# approach 1: run following and then run the output of this
    command
ps aux | grep -ie Xorg | awk '{print "sudo kill -9 " $2}'
# approach 2: run following
sudo killall Xorg
# approach 3: run following
sudo init 3
```

Start vitual display with:

```
sudo ls
sudo /usr/bin/X :0 &
```

You should see the virtual display starts successfully with the output as follows:

```
X.Org X Server 1.19.5
Release Date: 2017-10-12
X Protocol Version 11, Revision 0
Build Operating System: Linux 4.4.0-97-generic x86_64 Ubuntu
Current Operating System: Linux W5 4.13.0-46-generic #51-Ubuntu
    SMP Tue Jun 12 12:36:29 UTC 2018 x86_64
Kernel command line:
    BOOT_IMAGE=/boot/vmlinuz-4.13.0-46-generic.efi.signed
    root=UUID=5fdb5e18-f8ee-4762-a53b-e58d2b663df1 ro quiet
    splash nomodeset acpi=noirq thermal.off=1 vt.handoff=7
Build Date: 15 October 2017 05:51:19PM
xorg-server 2:1.19.5-0ubuntu2 (For technical support please see
    http://www.ubuntu.com/support)
Current version of pixman: 0.34.0
   Before reporting problems, check http://wiki.x.org
   to make sure that you have the latest version.
Markers: (--) probed, (**) from config file, (==) default
    setting,
   (++) from command line, (!!) notice, (II) informational,
   (WW) warning, (EE) error, (NI) not implemented, (??) unknown.
(==) Log file: "/var/log/Xorg.0.log", Time: Fri Jun 14 01:18:40
    2019
(==) Using config file: "/etc/X11/xorg.conf"
(==) Using system config directory "/usr/share/X11/xorg.conf.d"
```

If you are seeing errors, go back to "kill Xorg using one of following three way" and try another way.

Before running "Arena-Baselines" in a new window, run following command to attach a virtual display port to the window:

```
export DISPLAY=:0
```

17.3.2 Run Training

Create TMUX session (if the machine is a server you connect via SSH) and enter virtual environment:

```
tmux new-session -s Arena
source activate Arena
```

Continuous Action Space

List of games with continuous action space in *Arena*:

- ArenaCrawler-Example-v2-Continuous
- ArenaCrawlerMove-2T1P-v1-Continuous
- ArenaCrawlerRush-2T1P-v1-Continuous
- ArenaCrawlerPush-2T1P-v1-Continuous
- ArenaWalkerMove-2T1P-v1-Continuous
- Crossroads-2T1P-v1-Continuous
- Crossroads-2T2P-v1-Continuous
- ArenaCrawlerPush-2T2P-v1-Continuous
- RunToGoal-2T1P-v1-Continuous
- Sumo-2T1P-v1-Continuous
- YouShallNotPass-Dense-2T1P-v1-Continuous

Run the training commands, replace **GAME_NAME** with above games and choose proper **num-processes** (with **num-mini-batch** equivalent to **num-processes**) according to your machine,:

```
tmux new-session -s Arena
CUDA_VISIBLE_DEVICES=0 python main.py --mode train --env-name
    GAME_NAME --obs-type visual --num-frame-stack 4
    --recurrent-brain --normalize-obs --trainer ppo --use-gae
    --lr 3e-4 --value-loss-coef 0.5 --ppo-epoch 10
    --num-processes 16 --num-steps 2048 --num-mini-batch 16
    --use-linear-lr-decay --entropy-coef 0 --gamma 0.995 --tau
    0.95 --num-env-steps 100000000
    --reload-playing-agents-principle OpenAIFive --vis
    --vis-interval 1 --log-interval 1 --num-eval-episodes 10
    --arena-start-index 31969 --aux 0
```

Discrete Action Space

List of games with discrete action space in *Arena*:

- Crossroads-2T1P-v1-Discrete
- FighterNoTurn-2T1P-v1-Discrete
- FighterFull-2T1P-v1-Discrete
- Soccer-2T1P-v1-Discrete
- BlowBlow-2T1P-v1-Discrete
- Boomer-2T1P-v1-Discrete
- Gunner-2T1P-v1-Discrete
- Maze2x2Gunner-2T1P-v1-Discrete
- Maze3x3Gunner-2T1P-v1-Discrete
- Maze3x3Gunner-PenalizeTie-2T1P-v1-Discrete

- Barrier4x4Gunner-2T1P-v1-Discrete
- Soccer-2T2P-v1-Discrete
- BlowBlow-2T2P-v1-Discrete
- BlowBlow-Dense-2T2P-v1-Discrete
- Tennis-2T1P-v1-Discrete
- Tank-FP-2T1P-v1-Discrete
- BlowBlow-Dense-2T1P-v1-Discrete

Run the training commands, replace **GAME_NAME** with above games and choose proper **num-processes** (with **num-mini-batch** equivalent to **num-processes**) according to your machine,:

```
CUDA_VISIBLE_DEVICES=0 python main.py --mode train --env-name
    GAME_NAME --obs-type visual --num-frame-stack 4
    --recurrent-brain --normalize-obs --trainer ppo --use-gae
    --lr 2.5e-4 --value-loss-coef 0.5 --ppo-epoch 4
    --num-processes 16 --num-steps 1024 --num-mini-batch 16
    --use-linear-lr-decay --entropy-coef 0.01 --clip-param 0.1
    --num-env-steps 100000000 --reload-playing-agents-principle
    OpenAIFive --vis --vis-interval 1 --log-interval 1
    --num-eval-episodes 10 --arena-start-index 31569 --aux 0
```

You can also change other MARL algorithms instead of the PPO above to test the games you build.

17.3.3 Visualization

To visualize the learning curves for analyzing the training process with Tensorboard, run:

```
source activate Arena && tensorboard --logdir ../results/ --port
    8888
```

and visit http://localhost:4253 for visualization with tensorboard.

Acknowledgments We give special thanks to Yuhang Song, Prof. Zhenghua Xu, Prof. Thomas Lukasiewicz, et al., for their great contributions in the *Arena* project.

References

Balduzzi D, Garnelo M, Bachrach Y, Czarnecki WM, Perolat J, Jaderberg M, Graepel T (2019) Open-ended learning in symmetric zero-sum games. arXiv:190108106

Bansal T, Pachocki J, Sidor S, Sutskever I, Mordatch I (2018) Emergent complexity via multi-agent competition. In: ICLR. In: International Conference on Learning Representations. https://openreview.net/forum?id=Sy0GnUxCb

Cai Y, Daskalakis C (2011) On minmax theorems for multiplayer games. In: Proceedings of the twenty-second annual ACM-SIAM symposium on discrete algorithms. Society for Industrial and Applied Mathematics, Philadelphia

Heess N, Sriram S, Lemmon J, Merel J, Wayne G, Tassa Y, Erez T, Wang Z, Eslami S, Riedmiller M, et al. (2017) Emergence of locomotion behaviours in rich environments. arXiv:170702286

Hendtlass T (2004) An introduction to collective intelligence. In: Applied intelligent systems. Springer, Berlin,

Jaderberg M, Czarnecki WM, Dunning I, Marris L, Lever G, Castaneda AG, Beattie C, Rabinowitz NC, Morcos AS, Ruderman A, et al. (2018) Human-level performance in first-person multi-player games with population-based deep reinforcement learning. Computing Res Repository. http://arxiv.org/abs/1807.01281

Myerson RB (2013) Game theory. Harvard University Press, Cambridge

Singh S, Lewis RL, Barto AG (2009) Where do rewards come from. In: Proceedings of the annual conference of the cognitive science society

Singh S, Lewis RL, Barto AG, Sorg J (2010) Intrinsically motivated reinforcement learning: an evolutionary perspective. IEEE Trans Auton Ment Dev 2(2):70–82

Song Y, Wang J, Lukasiewicz T, Xu Z, Xu M, Ding Z, Wu L (2019) Arena: a general evaluation platform and building toolkit for multi-agent intelligence. arXiv:190508085