Griddly: A platform for AI research in games.

Chris Bamford,1 Shengyi Huang,2 Simon Lucas,1
1Queen Mary University
2Drexel University

Abstract
In recent years, there have been immense breakthroughs in Game AI research, particularly with Reinforcement Learning (RL). Despite their success, the underlying games are usually implemented with their own preset environments and game mechanics, thus making it difficult for researchers to prototype different game environments. However, testing the RL agents against a variety of game environments is critical for recent effort to study generalization in RL and avoid the problem of overfitting that may otherwise occur. In this paper, we present Griddly as a new platform for Game AI research that provides a unique combination of highly configurable games, different observer types and an efficient C++ core engine. Additionally, we present a series of baseline experiments to study the effect of different observation configurations and generalization ability of RL agents.

Background
Many prominent successes in research into artificial general intelligence (AGI) have emerged from creating agents that can achieve high scores in video games such as Atari 2600 games (Bellemare et al. 2013; Badia et al. 2020), custom toy game environments (Chevalier-Boisvert, Willems, and Pal 2018; Perez-Liebana et al. 2018), or using wrappers around popular video game such as Starcraft (Vinyals et al. 2019) DOTA 2 (Berner et al. 2019) and NetHack Learning Environment (Küttler et al. 2020).

Designing and implementing game environments to test the ability of different algorithms, such as Reinforcement Learning (RL), to reason, generalize and plan can be complex and time consuming. Even simple environments require implementing a number of common components such as rendering, game mechanics and optimization. A few solutions have been developed to abstract away the implementation details of environments and present researchers with a simplified interface to concentrate on building the specifics of environments to test their algorithms. For example the General Video Game Framework (GVGAI) (Perez-Liebana et al. 2018) provides a platform in which games can be defined by a Video Game Description Language (VGDL). VGDL contains a set of pre-defined instructions which can be combined together to create the mechanics of many games. The layout of levels can also be defined using a simple character-based 2D ASCII map. GVGAI is commonly used for research into general game playing (Torrado et al. 2018; Balla, Lucas, and Perez-Liebana 2020; Ye et al. 2020; Justesen et al. 2017), and procedural content generation (Drageset et al. 2019; Dharma, Togelius, and Soros 2020; Khalifa et al. 2020, 2016).

Another important factor in game environments is the rate at which game state can be produced for Deep Reinforcement Learning (DRL) algorithms, which generally run on Python. In many game environments the rate of data (rendered frames per second, or state representations per second) is relatively slow meaning long training times. Researchers can compensate for this problem by parallelizing the number of environments being used. This approach can be inefficient if the environments are not designed with integration with Python in mind. For example, GVGAI (Perez et al. 2019) run on the Java Virtual Machine (JVM) and have a huge communication overhead with Python through socket. Although this overhead can be mitigated through using Java Native Interface as done in gym-microcots (Huang and Ontañon 2020), the overhead still exists as opposed to games developed with C++ or Python.

In addition to the complexities involved in generating game mechanics and optimizing the game engine to allow fast prototyping during experimentation, the representation of game states used can also determine several factors during training such as how the agents generalize to unseen environment objects (Hill et al. 2019) and new levels (Ye et al. 2020; Balla, Lucas, and Perez-Liebana 2020). The representation of the states can also have a large impact on speed and memory usage during learning. RL from the pixels in rendered game frames requires significantly larger memory usage when using neural networks compared to if a simple one-hot or multi-class map representation of the game state is used.

The final consideration for game environments is that of how the agent or agents interact with the environment itself. Actions spaces differ across types of game environments, for example if the environment is an RTS game, the agent may need to provide coordinates in order to target certain actions, and available actions may differ across various units in the game. In single-player games, the agent may only need to provide a single value to control movement or rotation.
In multi-player games the environment needs an interface which provides the ability to control only certain units. In this paper we introduce “Griddly”, which provides a highly configurable and optimized platform for building grid-world games for artificial intelligence research. Game environments in Griddly, like GVGAI use a domain specific language known as Griddly Description YAML (GDY) which allows an unprecedented level of configurability in all of the key areas described above. Not only can GDY be used to create single player puzzle games like those in GVGAI, MiniGrid (Chevalier-Boisvert, Willems, and Pal [2018]) and other toy problems, but it can be used to create RTS games with multiple players, partial observability and complex resource systems. GDY also provides multiple in-built observation representations such as sprite based isometric rendering, simple shape-based tiles and minimal state vectors. In addition to the wide array of configurability that Griddly provides, the underlying Griddly engine is heavily optimized in both computational speed and memory usage by taking advantage of hardware accelerated rendering techniques.

To showcase Griddly, we provide a simple baseline of example experiments from 10 GVGAI-style games with various mechanics, configurations and observation representations. For each of the 10 games, we train a simple environment-size agnostic RL Agent on 5 different in-built levels, using three different observation configurations. In total this baseline results in 150 separate experiments. In addition to these 150, we provide 6 generalization experiments where we train an egocentric partial observability agent on three levels and evaluate its performance on two unseen levels.

Code for reproducing the experiments are made publicly available, so are the results and videos of each of these experiments. Full documentation, usage examples and tutorials for using Griddly are also provided.

The structure of the rest of this paper is as follows: Firstly we present a breakdown of the main configuration sections of Griddly’s description language (GDY), showing examples of creating simple game mechanics, how an environment can be set up for egocentric partial observability and how actions spaces can be defined. Secondly we show the results of the 150 baseline experiments that are trained per-level and the 6 generalization experiments. Thirdly we compare of the speed of generated states and memory usage between a number of game environments and their Griddly equivalent using GDY on the same hardware. Finally we discuss further features that we plan to add to Griddly and how Griddly can be used in future research.

**Griddly**

Griddly is an open-source project aimed to be a all-encompassing platform for grid-world based research. Griddly provides a highly optimized game state and rendering engine with a flexible high-level interface for configuring environments. Not only does Griddly offer simple interfaces for single, multi-player and RTS games, but also multiple methods of rendering, configurable partial observability and interfaces for procedural content generation.

Game mechanics are important building blocks of any game, they define the time and action dependent characteristics that turn a static image into an dynamic environment which can be used for understanding how fundamental logical understanding can help with much more complex tasks. As an example, the rules defined to allow an avatar to push a block one square at a time can be defined in a simple and deterministic way. This can be used to concentrate an AI agent learning to perform simple tasks such as collecting blocks in a specific area. In a physical world, this process would be significantly more complex. The block and the agent would be 3D, would have a high resolution view of the world, and there would be millions of variables to take into account, such as heat, friction, mass, forces of all the actuators of the agent etc. Using simplified game mechanics allows research to be undertaken on higher level concepts rather than have to take into account everything in the physical world.

Griddly is designed to provide an simple YAML interface, Griddly Description Yam (GDY) to allow many game mechanics to be defined in a simple way, without compromising on speed of rendering and mechanics. The next section will outline how the GDY schema can be used to create games.

**Griddly Description YAML (GDY)**

Griddly Description YAML (GDY) is a schema-oriented domain specific lanugage (DSL) which allows great flexibility
in creating grid-world environments. Any DSL requires a certain amount of pre-existing knowledge, however with industry standard configuration languages there are many tools available such as syntax highlighting, schema validation and linting, which can reduce the barrier to entry when writing DSLs. The use of YAML as a syntax allows schema validation (ISO 2020) to be used for syntax highlighting, validation and auto-completion. Many IDEs such as Visual Studio Code, IntelliJ and PyCharm support YAML validation out-of-the-box using JSON-schemas. This makes development of new games using Griddly simpler as the IDE will provide feedback on the game description’s syntax and structure. The main GDY configurable features of Griddly games are the **Environment**, **Objects** and **Actions**.

**Environment** The environment Section of the YAML contains configuration options for several high level concepts for a Griddly game. The three most important of these are the **Player**, **Termination** and **Levels** options. The player options define how the players will interact with the environment and which, if any, avatar object the player will control. Player partial observability can also be configured in the **Observer** subsection here using options such as: **RotateWithAvatar**, which causes the environment representation to rotate if the avatar rotates; **TrackAvatar**, which enables egocentric rendering (the agent is put at the center of the observation and **OffsetX** and **OffsetY** can be used to offset the agent from the center) and finally **Height** and **Width** which determine the height and width of the observable window.

Termination conditions, such as determining if the episode is complete, or determining the winner in a multiplayer game are also set in the environment section. Termination options can use any variable defined at a global level and also several special variables that allow calculations such as counts of specific objects in the environment. Levels are defined using strings of characters, the characters that are used are defined in the **Objects** configuration. An example of an environment configuration is shown below. An example of configured egocentric partial observability with an Isometric Renderer can be seen in figure 2.

```yaml
Environment:
  Name: sokoban
  TileSize: 24
  BackgroundTile: gvgai/newset/floor2.png
  Player:
    Observer:
      RotateWithAvatar: true
      TrackAvatar: true
      Height: 7
      Width: 7
      OffsetX: 0
      OffsetY: 2
      AvatarObject: avatar
    Termination:
      Win:
        - eq: [box:count, 0]

Levels:
  - |
    wwww
    .hA.w
    wwhw
    ..b.w
    whbb
    .www
    wwww
  - |
    wwww
    w.h....w
    w...hA.w
    w....w..w
    wwwb..w
    www...w
    wwwh....w
    wwww

Objects
  - Name: avatar
    Z: 2
    MapCharacter: A
    Observers:
      Sprite2D:
        Image: images/gvgai/oryx/knight1.png
```

**Actions** Instead of having a fixed set of actions, GDY allows the user to define any number of actions and how they will interact with other objects. Actions have the ability to modify the encapsulated variables of objects, for example; hit points, resources and possessions such as keys. Rendering information is also defined per-object and passed to the rendering engine at runtime. An example of the object definition for a single avatar object is given below:

```yaml
Objects:
  - Name: avatar
    Z: 2
    MapCharacter: A
    Observers:
      Sprite2D:
        Image: images/gvgai/oryx/knight1.png
```

**Actions** Instead of having a fixed set of actions, GDY allows the user to define any number of actions and how they will interact with other objects. Actions have the ability to modify the encapsulated variables of objects, for example allowing hit points to be modified, or adding and removing objects.

Actions in Griddly are defined in two parts, the **input mapping** and the **behaviours**. The input mapping maps a set of distinct integers to a *Description*, *OrientationVector* and *VectorToDest*. The *OrientationVector* and *VectorToDest* are parameters that can then be used by the *behaviours* to define how different objects react when the action is performed on them by another object. The object that is performing the action is referred to as the *source* object and object that is the target of the action is referred to as the *destination*. The following YAML snippet shows how the **input mapping** of a particular action is represented in GDY.

```yaml
InputMapping:
  Name: move
```
Figure 2: An example of the rendering of the Griddly environment spider nest with a specific player’s egocentric partially observable view that an agent will see during training (inset)

In order for this input mapping to translate into a full action, the behaviours of objects must be defined for this particular action. This is done by defining a list of commands that can happen to each object if they are the source or destination of the object. The source of an action is determined by the type of game that is being played. For example if the game is single player where the player controls a particular avatar, the source of the action is almost the avatar object (unless there are other actions which are automated such as random movement of enemies). In RTS-style games the source of the action is the object at the location selected by the player. An example of how Behaviours are defined in GDY is shown in below:

```plaintext
Behaviours:
- Src:
  Object: agent
  Commands:
  - rot: _dir
  Dst:
    Object: agent
- Src:
  Object: agent
Inputs:
1:
  Description: Rotate left
  OrientationVector: [-1, 0]
2:
  Description: Move forwards
  OrientationVector: [0, -1]
  VectorToDest: [0, -1]
3:
  Description: Rotate right
  OrientationVector: [1, 0]
Relative: true

Figure 3: Visualization of the three observers, Block, Sprite and Isometric configured on the "Spiders" environment. The 4th Vector observer does not have a natural visual representation.

When an action is performed, destination object is located by adding the value of VectorToDest to the source location. If the input mapping is defined with Relative:true, the VectorToDest and OrientationVector are calculated relative to the current orientation of the object.

Observers
As Griddly supports multi-player, single-player, and RTS style games, the format of the data in which the players (whether they are algorithms or humans), can be specified individually. There are four available observer types: Isometric sprite, block and vector. Isometric, sprite, block use hardware accelerated rendering through the Vulkan API (vul2020), an example of this is shown in Figure 3) vector rendering provides a lightweight class label representation of the grid. All observers support configurable partial observability and avatar tracking. Additionally multiple renderers can be used at the same time, meaning the game world can be observed separately from the agents. This allows rich demonstrations to be produced, even if the algorithms use the vector observer.
Potential Applications
GDY provides a flexible method of creating everything in a grid-world environment from game mechanics, to different ways of rendering the state. Griddly comes pre-loaded with 21 initial game examples, many of which are in this paper. We wish to highlight a number of research areas where using Griddly would be beneficial.

Environment Mechanics As games mechanics can be defined by combinations of simple instructions, many different game mechanics can be created. As an example, games that require random movement, ranged actions, directed movement (such as gravity) can be generated. Also resource systems can be easily defined by declaring variables that are local to objects or players, this means the possibility of complex games that require agents to collect particular items for specific tasks can be easily achieved.

Procedural Content Generation Due to the fact that the GDY language is YAML based, it is supported by most modern languages and there are many tools and libraries available to parse, edit and verify YAML files. The generation of game mechanics, levels and image assets can be performed by different projects regardless of language choices. The generated YAML can then be loaded by Griddly agnostically.

Baselines
To provide future experiments with a RL baseline, we ran two different types of experiment.

Firstly we chose 10 environments that have been ported from the GVGAI environment that are particularly difficult for RL algorithms to solve. Each environment contains 5 levels of different sizes and varying difficulty. We trained each level from each environment separately using three different observers as the observation space differs between different observers used. Experiments using Sprite and Block observers begin with a convolutional layer with kernel size and stride the same as the tile size. This effectively embeds each tile into tensor $\mathbb{R}^{32 \times W \times H}$ where $C$ is the number of channels, and $H$ and $W$ are the height and width of the game level in terms of tiles. Alternatively experiments using Vector observers embed the vector representations using a $1 \times 1$ kernel with 32 channels into the same resulting shape $\mathbb{R}^{32 \times W \times H}$. The embedded representation is then fed through the following layers: A convolutional layer with 32 input channels, 64 output channels, kernel size 3 and padding 1. Then a convolutional layer with 64 input and output channels, with kernel size 3 and padding 1. A global average pooling (GAP) layer is then used to reduce the network size to a linear layer of size 2048, this is then reduced by a further linear layer to a vector size 512. Single linear layers are used after this point for the various heads such as the actor and critic.

The global average pooling layer allows different $H$ and $W$ for different level sizes to output the same shape vector (in our case $2048$) (Balla, Lucas, and Perez-Liebana, 2020).

The action space of the fully observable environments consists of 5 actions do nothing (no-op), move up, move left, move down, and move right. In contrast, the partially observable setting the agent has access to 4 actions, do nothing (no-op), turn left, turn right and move forward.

Per-Environment
As shown in table 1 there are levels of certain environments that fail to gain any score. These environments are sometimes difficult to solve even for humans as they require precise planning and making single mistakes result in states that cannot be reversed. For example in the “Zenpuzzle” the agent gains a reward every time it moves over tiles of a certain colour, but once it has moved over these blocks, it can not move onto them again. This can cause the agent to get “stuck” surrounded by tiles that it has already passed. The agent can also easily stop itself from being able to reach certain tiles by blocking the path to them. Although the agent scores highly in the “Zenpuzzle” environment, it rarely covers “all” the tiles to reach the maximum score possible. In the game “Clusters” the agent must push coloured blocks into groups and scores a point each time a block is “grouped”. There are other obstacles in the environment that the agent must avoid pushing the blocks into. In the “Clusters” experiments the agent rarely learns to make a single group.

We test two partially observable versions of the games “Labyrinth” and “Zen Puzzle” to see if more general approaches, such as wall following strategies can be learned. As expected in "Labyrinth", a simple maze game with a reward at the destination, the agent performs better and can solve some of the mazes. We observe that in some of the games where a “wall following” approach can solve the maze, the agent learns this strategy. In other levels, the agent does not learn a strategy and fails to find the goal. It is also interesting to note that the full observable maze levels could not be learned by this method.
In the “Sokoban” environment, some levels are consistently solved. However, on other levels, the agent consistently fails to score a single point. This highlights that the structure of a level and the strategy required to solve the “Sokoban” levels can require different approaches to training, even when the mechanics of the game are consistent across levels.

### Generalization

The three training levels used in each experiment are 0, 1, and the evaluation levels are 1 and 3. Table 2 shows the results of these experiments. These experiments were also trained for 10M steps instead of the 1.28M used in the single environment experiments. In some levels, the agent is able to achieve some rewards, but was only able to solve a single unseen “Labyrinth” level out of the evaluation set. The result on the “Labyrinth” experiment trained per-level also shows that partial observability seems to be less of a challenge for the RL agent than having access to the whole level. The maximum episodic rewards for these environments can be seen in Table 1.

### Framework Comparison

In this section we provide two comparisons between several frameworks. The first comparison is a feature matrix showing the differences in features between Gridly and its closest grid-based relatives. The second comparison is between several games from popular frameworks that have been reimplemented using the GDY language.

#### Features

Table 3 shows how the features offered by Gridly compare with some other grid-based environments; since the paper is about Gridly it is presented to highlight what Gridly offers that other environments do not. Gridly does not currently provide access to a copyable forward model, but we plan to add this feature soon. Gridly is most closely related to GVGAI, but with various extensions to incorporate provide faster rendering and support for grid-based RTS games similar to μRTS (7).

Although ALE is not technically a grid-world, we’ve included it in the table for comparison. NetHack Learning En-

| Game         | Evaluation Level |
|--------------|------------------|
| Clusters     | 0.00 ± 0.00      |
| Cook Me Pasta| 4.00 ± 0.00      |
| Bait         | -0.09 ± 0.29     |
| Sokoban 2    | 0.00 ± 0.00      |
| Zen Puzzle   | 23.00 ± 0.00     |
| Labyrinth    | 0.00 ± 0.00 1.00 ± 0.00 |

Table 2: This table shows results of how well a simple neural network is able to generalize when trained on three levels and evaluated on two unseen levels. The agent can only see an egocentric partially observable view of the environments.

### Table 1

| Game         | Level | Max Reward | Vector | Block | Sprite |
|--------------|-------|------------|--------|-------|--------|
| Clusters     | 0     | 0.26 ± 0.34| 0.00 ± 0.00| 2.00 ± 0.00|        |
| Cook Me Pasta| 0     | 0.00 ± 0.00| 0.00 ± 0.00| 0.00 ± 0.00|        |
| Bait         | 0     | 5.00 ± 0.00| 5.00 ± 0.00| 5.00 ± 0.00|        |
| Sokoban 2    | 0     | 4.90 ± 0.00| 4.90 ± 0.00| 4.90 ± 0.00|        |
| Zen Puzzle   | 0     | 2.00 ± 0.00| 2.00 ± 0.00| 2.00 ± 0.00|        |
| Labyrinth    | 0     | 1.00 ± 0.00| 1.00 ± 0.00| 1.00 ± 0.00|        |
| Zen Puzzle   | 0     | 2.00 ± 0.00| 2.00 ± 0.00| 2.00 ± 0.00|        |

Table 1: This table shows the results of training 5 levels of 10 GVGAI environments that have been ported to the Gridly platform using GDY. Each level from each environment is trained for 1.28 million steps and the average episode reward of the final 100 episodes is reported for each observer used. (Vector, Block and Sprite). In most cases the results are consistent across all the representations, but due to the unstable nature of RL based on random starting seeds, some errors in consistency are expected. The two environments marked with ? are configured with egocentric partial observability included it in the table for comparison. NetHack Learning En-
Efficiency

As the focus of the Griddly Engine is currently to improve the data rate of RL in grid-world environments, a benchmark comparison of the available python gym interfaces for some of most popular grid-based environments is shown in Table 4. The benchmark consists of running the original environment and the equivalent Griddly version with a random agent for 10000 frames and calculating the average frames-per-second (FPS) of the rendered states and memory usage. Rendering the pixels of the environments is the most demanding method of producing game states, so it provides a useful bottleneck to test. Additionally we compare the vectorized versions of the states produced by the game engines if available. The games used for the test are as follows: GVGAI - Sokoban, MiniGrid - FourRooms, gym-microrts (Huang and Ontañón 2020) - BWDistantResources32x32

| Platform  | FPS (Rendered) | FPS (Vector) | Memory (MB) |
|-----------|----------------|--------------|-------------|
| Griddly   | 2901           | 30150        | 106         |
| GVGAI     | 18             | -            | 365         |
| Griddly   | 2317           | 40174        | 106         |
| Minigrid  | 222            | 1024         | 106         |
| Griddly   | 2722           | 11574        | 313         |
| gym-microrts | 195          | 1294         | 313         |

Table 4: Speed and memory footprint of Griddly compared to similar environments. Note that the others are accessed via the Open AI Gym interface, which in the case of GVGAI reduces the speed significantly. In each double-row the Griddly entries are for the same or similar game running on each of the platforms.

Environment (NLE) is built around the classic NetHack Roguelike game, and although just one grid-based game, it offers great variety due to procedural level generation. Although not included in the table, ProcGen is similar to ALE in scope, but offers endless variety through procedural level generation.

Future Work

While Griddly provides a fast and flexible platform for research such as RL, there are algorithms that require a copyable forward model of the environment to effectively “try out” potential actions in a copy of the real environment. These algorithms, for example Monte-Carlo Tree Search and Evolutionary algorithms are currently not supported by the Griddly Framework. It is the authors intention to implement a copyable forward model in a future version.

Additionally, as Griddly is written in C++ there are many languages that can communicate with the underlying native implementation. Currently a Python interface is supported for ease of use with openai gym, tensorflow and pytorch. Additional libraries are planned for example a Julia wrapper is currently under development.

Conclusion

In this paper we introduce Griddly, a new, highly efficient framework that allows grid-world games to be easily created using a very flexible description language (GDY) which allows flexibility in desired game mechanics, provides configurable interfaces for various observation, action spaces, level design, and reward functions. This level of flexibility enables a wide range of research possibilities including game playing agents using RL, and level and game design using procedural content generation. The YAML-based GDY language also reduces barriers in language and framework choices as YAML is a widely supported standard.

We provide two sets of RL experiments to make a baseline for future work. The first experiment trains an RL agent separately on different levels of several games with various configurations of observability made possible with Griddly. The second experiment trains a single agent on a few levels of a particular game and evaluates on a few unseen levels. In total a baseline of 156 experiments is given. Both experiment sets show that although the basic RL agents converge, sometimes to the maximum score, the majority of the environments fail to be solved and fail to generalize across new environments. We believe this provides a strong incentive for future experimentation.
References

2020. Json Schema. https://json-schema-everywhere.github.io/yaml

2020. Vulkan. https://www.khronos.org/vulkan

Badia, A. P.; Piet, B.; Kapturowski, S.; Sprechmann, P.; Virvitskiy, A.; Guo, D.; and Blundell, C. 2020. Agent57: Outperforming the Atari Human Benchmark. arXiv URL https://arxiv.org/abs/2003.13350

Ball, M.; Lucas, S. M.; and Perez-Liebana, D. 2020. Evaluating Generalisation in General Video Game Playing. arXiv URL https://arxiv.org/abs/2005.11247

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: an evaluation platform for general agents. Journal of Artificial Intelligence Research 47: 253–279. ISSN 1076-9757. doi:10.1613/jair.3912. URL https://jair.org/index.php/jair/article/view/10819

Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Debiak, P.; Dennison, C.; Farihi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; Józefowicz, R.; Gray, S.; Olsson, C.; Pachocki, J. W.; Petrov, M.; de Oliveira Pinto, H. P.; Raiman, J.; Salimans, T.; Schlatter, J.; Schneider, J.; Sidor, S.; Sutskever, I.; Tang, J.; Wolski, F.; and Zhang, S. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. ArXiv abs/1912.06680

Burda, Y.; Edwards, H.; Storkey, A.; and Klimov, O. 2018. Exploration by Random Network Distillation. arXiv URL https://arxiv.org/abs/1810.12894

Chevalier-Boisvert, M.; Willems, L.; and Pal, S. 2018. Minimalistic Gridworld Environment for OpenAI Gym. https://github.com/maximecb/gym-minigrid

Dharna, A.; Togelius, J.; and Soros, L. B. 2020. Co-Generation of Game Levels and Game-Playing Agents. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment URL https://www.aaai.org/.ojs/index.php/AAIDE/article/view/7431

Drageset, O.; Winands, M. H. M.; Gaina, R. D.; and Perez-Liebana, D. 2019. Optimising level generators for general video game AI. In 2019 IEEE Conference on Games (CoG). 1–8. IEEE. ISBN 978-1-7281-1884-0. doi:10.1109/{CIG}.2019.8847961. URL https://ieeexplore.ieee.org/document/8847961

Hill, F.; Lampinen, A.; Schneider, R.; Clark, S.; Botvinick, M.; McClelland, J. L.; and Santoro, A. 2019. Environmental drivers of systematicity and generalization in a situated agent. arXiv URL https://arxiv.org/abs/1910.00571

Huang, S.; and Ontaño, S. 2020. Action Guidance: Getting the Best of Sparse Rewards and Shaped Rewards for Real-time Strategy Games.

Justesen, N.; Bontrager, P.; Togelius, J.; and Risi, S. 2017. Deep Learning for Video Game Playing. arXiv URL https://arxiv.org/abs/1708.07902

Khalifa, A.; Bontrager, P.; Earle, S.; and Togelius, J. 2020. PCGRL: Procedural Content Generation via Reinforcement Learning. arXiv URL https://arxiv.org/abs/2001.09212

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J. 2016. General video game level generation. In Friedrich, T., ed., Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16. 253–259. New York, New York, USA: ACM Press. ISBN 9781450342063. doi:10.1145/2908812.2908920. URL http://dl.acm.org/citation.cfm?doid=2908812.2908920

Küttler, H.; Nardelli, N.; Miller, A. H.; Raileanu, R.; Seltavici, M.; Grefenstette, E.; and Rocktäschel, T. 2020. The NetHack Learning Environment. arXiv URL https://arxiv.org/abs/2006.13760

Perez, D.; Liu, J.; Abdel Samea Khalifa, A.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2019. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. IEEE Transactions on Games 1–1. ISSN 2475-1502. doi:10.1109/[(TG).2019.2901021. URL https://ieeexplore.ieee.org/document/8664126

Perez-Liebana, D.; Liu, J.; Khalifa, A.; Gaina, R. D.; Togelius, J.; and Lucas, S. M. 2018. General Video Game AI: a Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. arXiv URL https://arxiv.org/abs/1802.10363

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. arXiv URL https://arxiv.org/abs/1707.06347

Torrado, R. R.; Bontrager, P.; Togelius, J.; Liu, J.; and Perez-Liebana, D. 2018. Deep reinforcement learning for general video game AI. In 2018 IEEE Conference on Computational Intelligence and Games (CIG). 1–8. IEEE. ISBN 978-1-5386-4359-4. doi:10.1109/{CIG}.2018.8490422. URL https://ieeexplore.ieee.org/document/8490422

Vinylas, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; Oh, J.; Horgan, D.; Kroiss, M.; Danihelka, I.; Huang, A.; Sifre, L.; Cai, T.; Agapiou, J. P.; Jaderberg, M.; Vezhnovets, A. S.; Leblond, R.; Pohlen, T.; Dalibard, V.; Budden, D.; Sulsky, Y.; Molloy, J.; Paine, T. L.; Gulcehre, C.; Wang, Z.; Pfaff, T.; Wu, Y.; Ring, R.; Yogatama, D.; Wünsch, D.; McKinney, K.; Smith, O.; Schaul, T.; Lillicrap, T.; Kavukcuoglu, K.; Hassabis, D.; Apps, C.; and Silver, D. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature 575(7782): 350–354. ISSN 0028-0836. doi:10.1038/s41586-019-1724-z. URL http://www.nature.com/articles/s41586-019-1724-z

Ye, C.; Khalifa, A.; Bontrager, P.; and Togelius, J. 2020. Rotation, Translation, and Cropping for Zero-Shot Generalization. arXiv URL https://arxiv.org/abs/2001.09908