Universiteit
Leiden
The Netherlands

Bachelor Computer Science

Godapt: Introducing general video game AI

to the Godot game engine.

Thijs Wim Dekker

Supervisors:
Matthias Müller-Brockhausen & Mike Preuss

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

16/07/2025

Abstract

General Video Game Playing (GVGP) is a rich and evolving area of research, posing significant theoretical and practical challenges. Previously explored methods to achieve general AI in games have oftentimes been pestered with limitations, such as frameworks being built in custom languages, or the AI agent being restricted to a limited set of video games. This thesis presents the results of a new approach to GVGP agent creation, namely an integration into the already existing Godot video game engine. It looks at both the possibility of such an integration and lays the foundation for GVGP tools by introducing a program we developed for this purpose: Godapt.

Where other projects before us have made reinforcement learning frameworks for the Godot game engine by adding nodes within the editor, we have explored the possibility of external integration. This allows our framework to interact with games made within Godot, without the user having to change anything within the game. In this research we empirically show that it is possible to adapt Godot in a way that enables the creation of both learning and planning algorithms within the engine. All of the code, including the benchmark agents, is available in the public domain.

Contents

1 Introduction

The idea and interest in machines outperforming humans in games has been around for a long time. One of the earliest instances of such a machine can be traced back to the 1700s, where the "Mechanical Turk" dominated its opponents in chess. It was discovered, however, that this automaton was operated by a chess master from within the machine [Sch99]. Whilst the Mechanical Turk turned out to be a fake, the curiosity behind it was not, as would be shown by the rise of chess engines in the 20th century. The swift improvement in both the quality of software and hardware led to the iconic match between the chessbot Deep Blue and the then reigning chess world champion Garry Kasparov[CHhH02]. Deep Blue won the match, becoming the first ever chessbot to defeat a world champion. Whilst the improving chess engines garnered a lot of attention, scientists also broadened their vision for even more complex environments.

With the rise of computers came new frontiers. Video games became ever more prevalent in a computer-oriented society. This led to some scientist focusing their attention towards developing AI players in video games. There are two paths to improving an AI agent in games: make the agent perform as good as possible in a single game, or make the agent perform as good as possible over several different games. The latter is a more niche field of research, namely the field that looks into General Video Game Playing (GVGP). A prominent project utilizing a framework to help develop general agents was the Arcade Learning Environment (ALE)[BNVB13]. The ALE framework was designed to develop general AI agents capable of playing a wide range of Atari 2600 games, significantly advancing research in general AI creation.

A problem that the ALE and many of its successors ran into, was that such a framework severely limits the scope and capabilities of AI agents made specifically for it. One of ALE's problems is that the set of games is finite and limited, which makes it so the agents will never be able to play video games that are not within its dataset. A more modern framework, widely adopted by researchers, is the General Video Game Artificial Intelligence (GVGAI) framework. The GVGAI framework is built upon its own programming language, the Video Game Description Language (VGDL), as proposed in *Towards a Video Game Description Language*[ELL+13]. Having a language instead of a set amount of games solved the issue of having a finite test set. Due to the consistent and structured nature of the VGDL, the GVGAI framework became popular with scientists researching general AI. This fact, combined with the frequent GVGAI competitions using the GVGAI framework, has led to new methods being explored and numerous articles written for GVGAI research. Notable examples include deep learning by interfacing to the OpenAI gym environment by Torrado et al.[TBT+18], or the exploration of diversifying the objectives of general AI heuristics by Guerrero-Romero et al.[GRLPL17].

The rise in popularity of general AI research demands both better AI agents and better development frameworks. The GVGAI framework has proven itself to be a popular choice for research, but even such a well-established framework leaves room for improvement. Many aspects of the framework could be updated to allow for other more complex situations, such as multi-agent or multi-player GVGAI, or even automatic game design, as discussed by Perez-Liebana[PLLG+19]. The GVGAI framework might have solved the problem of having access to a finite amount of games, but there is still the inherent issue with the VGDL language. It restricts the agents into playing games only made with VGDL, which excludes all games created using more common methods, such as libraries or game engines.

1.1 Problem statement

This thesis seeks to explore the feasibility of developing a general video game AI framework utilizing existing games. In order to achieve this, the Godot game engine has been chosen, due to its popularity and the open-source nature of the software. The research question for this thesis is as follows: *Is it possible to make an AI framework built within Godot similar to GVGAI in which agents can play games using both planning and learning methods?* To answer this question, the research has been split up into four distinct steps. The first goal is to find out whether it is possible to run a random agent in a Godot project, without having to edit anything within the pre-existing game.

Second, the feature of Reinforcement Learning support will be explored, by recreating the frozen lake environment from openAI's Gymnasium library[1] [BCP+16]. Data from Q-learning agents in both the framework and the Gymnasium environment will be compared to ensure that the framework produces the same results.

Lastly, the possibility of making a Forward Model (FM) and using Monte-Carlo Tree Search (MCTS) within the framework needs to be implemented. This is due to the fact that some general AI implementations rely on the possibility of MCTS, such as the Self-Adaptive MCTS agents made by Sironi et al.[SLPL+18]. While other approaches such as Rolling Horizon Evolutionary Algorithms (RHEA) are also an often used approach to AI agents, they will not be explored within this research in regards to the relatively small scope of a bachelor's thesis.

By creating an external framework in Godot that can both facilitate the creation of Reinforcement Learning agents and supports the use of MCTS within a FM, the goal of showing the possibility of a general video game AI framework utilizing existing Godot games within the test and training sets will have been achieved.

1.2 Thesis overview

This bachelor thesis, created within LIACS and supervised by Matthias Müller-Brockhausen and Mike Preuss, begins with an exploration of the underlying problem here in section 1. Section 2 discusses related work; Section 3 includes the definitions and relevant background information; Section 4 goes into the design and inner workings of the Godapt framework; Section 5 describes the experiment and its outcome; Section 6 concludes, goes into further detail about the shortcomings of the software and presents possible future research.

[1] https://gymnasium.farama.org/environments/toy_text/frozen_lake/

2 Related Work

Many different wrappers and bindings have been developed for most major game engines, such as Unity, Unreal Engine and Godot. Most of these solutions convert the game environment to an OpenAI Gym environment or a similar construction to be able to access it through Python. While this is an apt solution for Reinforcement Learning alone, it does not take into account planning through algorithms such as MCTS, which require a separate simulation of the game world; a feature not yet implemented within the aforementioned engines. We will be looking into one of the Reinforcement Learning projects for Godot, namely GymGodot, after discussing the project that inspired this research: GVGAI.

2.1 GVGAI

The General Video Game Artificial Intelligence (GVGAI) framework[PLLG+19] is a well-known platform built upon the Video Game Description Language (VGDL)[ELL+13] and introduced as a means to evaluate general game-playing agents over a diverse set of games. Written in Java, GVGAI offers many 2D environments for researchers to test or train agents.

The framework can be seen as "split in two". Firstly there is the *planning* track in the original framework that focuses on letting agents simulate a future environment: a so-called Forward Model (FM) [PLLG+19]. AI agents can use algorithms such as MCTS or RHEA to make probability-based decision with the knowledge they can extract from the FM. In 2017, a new interface was implemented on top of the original GVGAI framework, in which the agents were restricted from accessing the FM. This created a *learning* environment where Reinforcement Learning agents could be trained and tested by continuously replaying the games available. This learning environment is where the well-known GVGAI competitions started.

Despite its widespread use, the original implementation of GVGAI has several limitations. Not all of them will be discussed here, but an important limitation that is relevant to this research is its dependency on VGDL. VGDL limits the flexibility of the logic by enforcing that the game stays grid-based and avatar-centric. Having such rules makes it easier for agents to communicate with the framework, as discrete worlds and other such limitations carry over into every single game, making it so that there is an overlap of rules between games. While this makes it easier to learn these games as a general agent, it also severely limits how "general" the agent truly is, as it will never come into contact with aspects such as continuous motion or physics. This ends up limiting the variation of games that are available, and thus limits the learning scope of agents interfacing with GVGAI.

The GVGAI book mentions a possible solution, saying: "This can be complemented with adding an integration with other systems. Different general frameworks like like OpenAI Gym [BCP+16], ALE [BNVB13] or Microsoft Malmö [JHHB16] already count on a great number of games (single, multi-player, model free and model based). Interfacing with these systems would increment the number of available games which all GVGAI agents could play via a common API." (Perez-Liebana, 2019, [PLLG+19]). This solution would alleviate some of the problems that come from having a framework built upon VGDL, but the solution that the research in this paper proposes is to avoid VGDL entirely in order to acquire a broader amount of games with a bigger set of differences between them, giving agents more freedom.

2.2 GymGodot

A popular approach to Reinforcement Learning agent integration in game engines is to set up communications between the game engine and a Python OpenAI Gym environment. It essentially makes it so that there is an agent and environment within the engine that can be accessed through an API within Python. One of the projects that has been implemented in such a way within Godot is the GymGodot project.

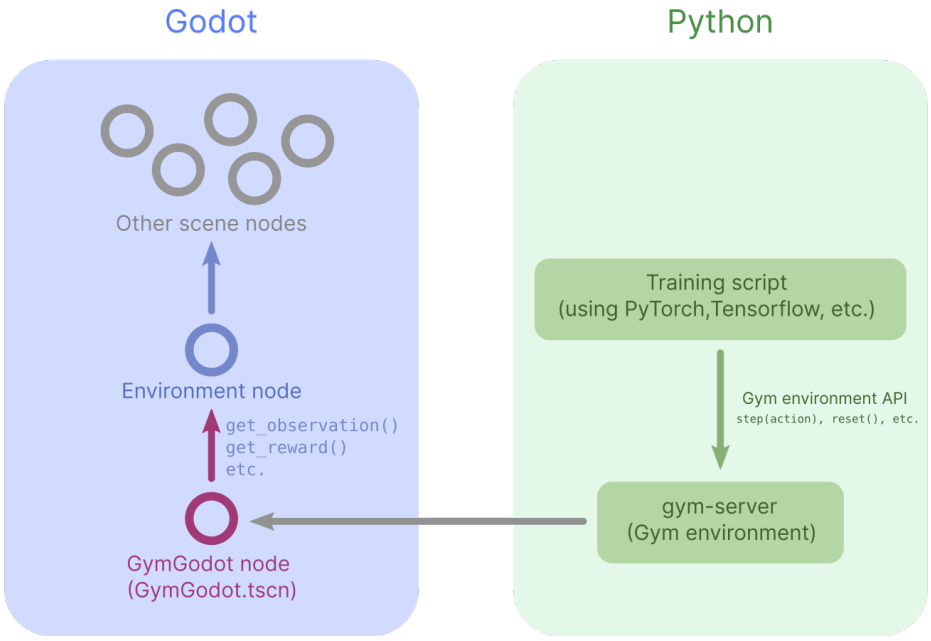


Figure 1: Overview of the GymGodot framework communication (from Hugo Tini, 2021, https://github.com/HugoTini/GymGodot)

The way GymGodot achieves this is through a WebSocket. The custom ServerEnv class sets up the server side in Python and the custom WebSocketClient class uses the built-in WebSocketPeer on the Godot side to connect as a client. The framework is used similarly to a normal implementation of the OpenAI Gym environment on the Python-side, with the a similar API. Behind the scenes, however, the Server Environment sends these function calls through the WebSocket to Godot, where an Agent node and an Environment node process these requests and send the results back through the WebSocket to the Python side, as seen in figure 1. A big restriction such an implementation imposes is the fact that the game's scene needs to be changed in order to facilitate the GymGodot and Environment Nodes, meaning the game itself needs to be edited in order to be used by GymGodot.

This framework enables the creation of RL agents that can make use of the plethora of Python libraries designed to aid with RL, such PyTorch and Tensorflow, while also utilizing Godot and its potentially infinite set of games. While this framework is not necessarily designed to be used by *general* AI agents, it can easily be used this way by swapping out the Godot side each time the agent wants to play a different game. An project with a similar approach as GymGodot is Avalon[AFF+22], a 3D video game environment and benchmark designed for Reinforcement Learning research. Avalon shows the viability and practical utility of the WebSocket approach.

What this framework lacks in order to facilitate proper creation of general agents, is a planning

track. In order the use algorithms such as MCTS or RHEA, a simulation of the environment and its future state, separate of the original game environment, needs to be created. Currently, the Godot side of this framework only facilitates a single SceneTree, as is the case in an unedited build of Godot. A hard limit of only one SceneTree makes it impossible to create a simulation of the environment parallel to the current game environment, thus making it an unfeasible option for agents that use planning instead of learning.

3 Background Information

This section covers the technical aspects and definitions behind this research. This paper assumes a basic understanding of Reinforcement Learning (RL), with all RL definitions used being based upon those found in *Reinforcement learning: An introduction (2nd ed.)* by Sutton and Barto, 2020 [SB20]. However, this section will quickly dive into the basics of Monte-Carlo Tree Search (MCTS) 3.1, as this is an integral part of the final steps of this research. For those without experience with the Godot game engine and its specifics, there will be an overview of its general layout and inner workings in section 3.2.

Although the reader's general knowledge of AI in games is implied, there are distinctions to be made between General Game Playing (GGP) and General Video Game Playing (GVGP). In GGP, the game, often a turn-based board game, is defined using a declarative approach that specifies the game's logic. In the article *General video game playing*, the field of General Video Game Playing (GVGP) was introduced. GVGP was described by Levine et al. as follows: "In GVGP, computational agents will be asked to play video games that they have not seen before. At the minimum, the agent will be given the current state of the world and told what actions are applicable. Every game tick the agent will have to decide on its action, and the state will be updated, taking into account the actions of the other agents in the game and the game physics" [LCE+13].

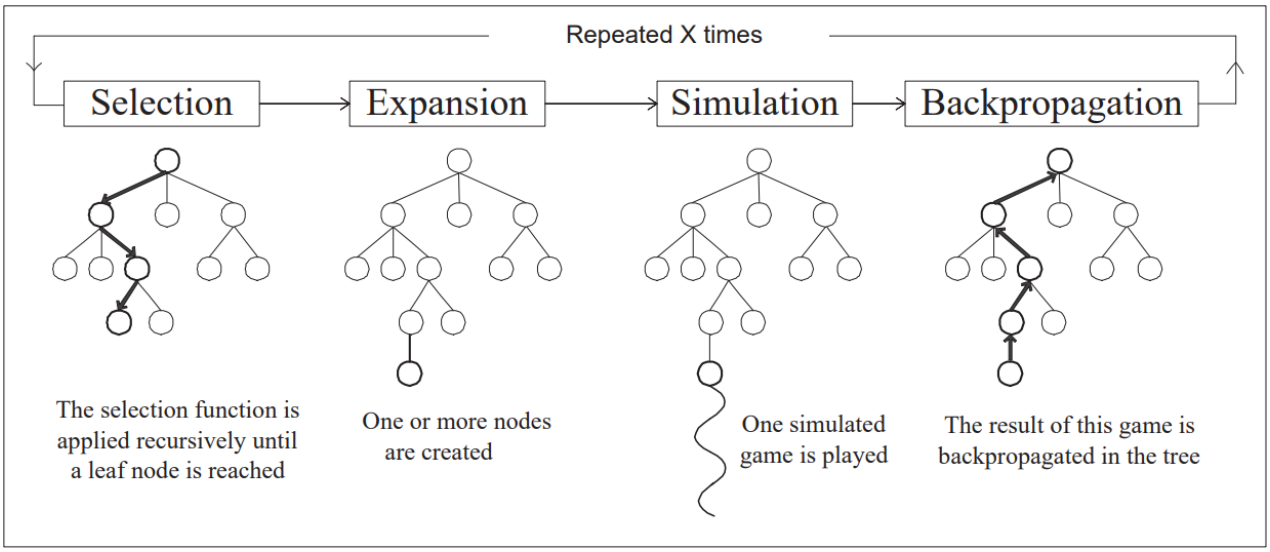


Figure 2: Outline of a Monte-Carlo Tree Search (from Chaslot et al., 2008, p.3[CWU+08])

3.1 Monte-Carlo Tree Search

This section is a summarized version of "Section 2: Monte-Carlo Tree Search" in the article *Progressive Strategies for Monte-Carlo Tree Search* [CWU+08] and covers the basics needed to understand it.

MCTS is a best-first search method that is compromised of four different stages, as can be seen in figure 2. The algorithm begins with the **selection** stage, where it starts out in the root node R (the current state, in terms of games). From the root node the algorithm recursively visits children nodes N, selecting them based off of a selection function. This thesis does not dive into complex heuristics and exclusively uses random selection during the selection stage. Once the selection stage reaches a leaf node L, it will start the **expansion** stage. The current leaf node will be expanded by storing one or more of its children into memory. From this point the **simulation** stage starts, where moves will be selected until either the end of the game (often the case when simulating a zero-sum game), or until a set amount of moves or time is exceeded. This thesis will not use anything more complex than random selection during the simulation stage. Once the playout is done, the **backpropagation** stage starts. The backpropagation takes the result of the simulated game k and propagates it backwards through all the visited nodes starting from the selected leaf node L, while also updating the visit count n_N of each node. The value of a node is computed by taking the average value v_N of all simulated games that have used node N, using the formula:

$$v_N = \frac{\Sigma_k R_k}{n_N}$$

These four stages get repeated X amount of times or until a certain time threshold has been exceeded. Once this is the case, the program will play the move corresponding the the "best" child of the root node. There are numerous ways to define which child is best using the value and visit count of each node. This thesis will be using the *Max child*: the child with the highest value.

An example of pseudo-code based off of Chaslot et al., 2008, p.4 [CWU+08] can be found in algorithm 1. In this pseudo-code, *ST* stand for the set of nodes that are loaded into memory and thus are part of the search tree. The SELECT(*Node N*) function call is the selection function that decides which children to choose during the selection stage. EXPAND(*Node N*) chooses one of the node's children and expands it. The function PLAY_SIMULATED_GAME(*Node N*) does a single playout starting from the child node chosen during the expansion stage and returns the result. This corresponds to the simulation stage. BACKPROPOGATE(*Float/Integer R*) calculates the new value for the node using the earlier mentioned formula. The function PICK_BEST_CHILD(*Node N*) picks the best child after all four stages have stopped looping.

Algorithm 1 Monte-Carlo Tree Search

```
1: function MCTS(root_node)
2:     while has_time do
3:         current_node ← root_node
4:         while current_node ∈ ST do
5:             last_node ← current_node
6:             current_node ← SELECT(current_node)
7:         end while
8:         last_node ← EXPAND(last_node)
9:         Result ← PLAY_SIMULATED_GAME(last_node)
10:        while current_node ∈ ST do
11:            current_node.BACKPROPOGATE(Result)
12:            current_node.visit_count += 1
13:            current_node = current_node.parent
14:        end while
15:    end while
16:    return PICK_BEST_CHILD(root_node)
17: end function
```

3.2 Godot

Godot is a cross-platform game engine that supports both 2D and 3D game creation. Godot's free and open-source nature, supported by an MIT license, makes it an excellent choice for game development and research. An understanding of the underlying structure of Godot is crucial to understanding the design choices and results of this thesis. This section will give a rundown of the important factors of Godot that have shaped the decision making and design of the research.

While Godot supports `C#` directly and other languages through its GDNative feature, the Godot editor mainly relies on its built-in programming language: **GDScript**. GDScript is a high-level, dynamically typed programming language, designed with integration with the `C++` engine in mind. The language is oftentimes compared to Python, with the two languages sharing very similar syntax. While the two are similar, GDScript lacks the vast number of libraries that Python has access to, making it less suited for problems that would be solved easier using a pre-made library.

Godot uses a unique structure for its engine. Every game is built entirely from **Nodes**, the building blocks of Godot. Every possible thing in Godot inherits from this node class, with many different types available. User-made scripts can be attached to these nodes, giving them unique behaviors, with nodes always having the functions `_ready()`, called when this node and all of its children have entered a new tree, `_process()`, called each frame, and `_physics_process()`, called for each physics frame, along with a handful of other functions that handle features like inputs. These nodes are arranged in a tree called a **Scene**, with nodes having parent-, sibling-, or child-nodes. These trees or their sub-trees can be saved as an individual **PackedScene**, functioning similar to "prefab" objects in other engines. Scenes also make up the environments and levels of your game.

At the heart of the GDScript side of the engine lies the **MainLoop** class, an abstract class designed to control the game loop of Godot. The **SceneTree** inherits this from the MainLoop, but adds in all

the utility needed to manage a Scene and the Nodes within it, making it the default implementation of the MainLoop in most use cases. The SceneTree and the classes that inherit from it are the main connection from the `C++` engine to the user-written GDScript side of things. The loop on the `C++` side calls functions such as `_process()` and `_physics_process()`. The SceneTree handles its logic, but also makes sure that all the Nodes in its current scene have their respective process functions called as well. An important detail to note here is that the MainLoop class, and thus the SceneTree through inheritance, is defined as a singleton, meaning that in the current release of Godot only *one* instance of such a class is allowed, otherwise undefined behavior might occur. This attribute complicates the creation of a secondary Mainloop, making the development of simulations complicated.

4 Godapt

In order to answer the research question of this thesis, a framework integrated with Godot had to be built. The final result of this project is the software called Godapt: an open-source framework developed in GDScript using a custom-built version of Godot. While this means that the project technically does not run on a normal version of Godot, the custom-built version of Godot makes sure that games built with the original engine also run within this custom version, which is in line with one of our design philosophies: Any arbitrary game created within Godot version 4 must be compatible with Godapt. The code behind the framework itself and the custom Godot build can be found on their respective Github pages.

This section is split up into several distinct subsections that each touch on a different part of the research. Section 4.1 is meant as user manual that teaches the reader how to use the software themselves, from the setup to interacting with the API, meaning this section is not of importance to the results of the research itself. Section 4.2 explains how to software is built and structured and explores why these design choices were made. It goes into further detail about which options were considered and why some of these approaches deemed infeasible or unviable. Lastly, section 4.3 looks into the current framework and its limitation. It mainly touches on why these limitations exist, whereas section 6 goes into further detail on how to fix some of these issues for possible further research.

4.1 Functionalities

Godapt runs on a Godot executable compiled for x64 Windows and uses a shell script to start up, meaning a x64 Windows operating system along with an installation of Bash is required in order to use the program. While Linux is not directly supported, it is possible to compile a Linux version of the custom Godot engine by cloning it from the Github page[2], compiling it using `scons platform=linux target=template_debug`, putting the acquired executable in the `\GodotBuilds` folder and linking it in the `RunFramework.sh` file.

[2] https://github.com/T-Dekker/godot

4.1.1 Setup

This setup guide assumes the user already has a Godot game set up in which they would like to use Godapt. These first steps explain how to set up the bare minimum of Godapt within your own Godot project.

Basic setup:

1. Create a folder named *Godapt* within your existing Godot project, on the same level as your `project.godot` file.
2. Clone the Godapt Github repository[3] into the newly made folder, either by downloading it directly or by using `git clone https://github.com/T-Dekker/Godapt`.
3. If an agent GDScript file has already been created, move it anywhere into the project. If not, create a GDScript file anywhere in your project that extends the Agent class, by starting the script with `class_name CustomAgentName extends Agent`.
4. Open up the project in your preferred version of Godot. This ensures Godot registers the new files, otherwise these would not be accessible during runtime.

After these steps, Godapt is successfully linked within your Godot project. The following steps guide you on your way to make your first agent interact with the game.

First time running:

1. Set up the desired settings within the `settings.JSON` file. For more information on each variable's function, please refer to section 4.1.2.
2. Run the project through the framework by running the `RunFramework.sh` file, using the command line options.
   (a) `-w` runs the project in windowed mode. Without this command, the program runs headless. This option is currently not supported in combination with MCTS, due to limitations that will be discussed in section 4.3.
   (b) `-v` runs the project in verbose mode. This is no different than Godot's own verbose mode.
   (c) `-2` runs the project using the most recent version of Godapt. This is a less stable version of Godapt that allows for FM creation.

4.1.2 Settings

Godapt comes with a `settings.JSON` in which several variables can be set. Some of these variables have to be set in order for the framework to function as intended. An explanation for these settings can be found in table 2. Some settings are purely optional and only intended to help change variables within the agent without having to hard-code them. While possibly anything can be sent to the agent this way, these settings were meant for certain purposes, which can be found in table 1.

[3] https://github.com/T-Dekker/Godapt

Table 1: Overview of optional settings

Setting	Type	Default	Description
`train`	Boolean	true	Boolean sent to the agent. Meant to select a training or playing mode.
`load_data_path`	String	`""`	A string sent to the agent. Meant to indicate the file path of an RL agent's data, so it can load it before running.
`save_data_path`	String	`""`	A string sent to the agent. Meant to indicate the file path of an RL agent's data, so it can save it after running.

4.1.3 Documentation

This section goes over how to properly make use of Godapt's features, which includes the Agent class's functions that should be overwritten and how to set up a scene. This section will not go into detail about the inner workings of the framework, for that, refer to section 4.2.

Because of Godapt's design, it does not have an API in the original sense. It can not be called through code the same way libraries are, but instead it has a loop for itself where you can insert parts of your own code. The main loop in the framework calls functions within the user made Agent class. By overriding these functions, the user can interact with the game while getting a constant feed of information, thus making it possible to develop agents within Godot.

The Agent class follows the rules of a classic RL agent. After initializing, it continually stays in a loop where it receives a reward r_t and a state s_t from the environment, with which it can make a choice for its next action a_t and update its own data. The loop itself is inaccessible in Godapt, but the functions on the custom agent class are called accordingly. In table 3 are all the functions available and how to utilize them.

For selecting actions, the Agent class has been provided with the variable `actions: Array[StringName]` that contains all the inputs that were provided in the `settings.JSON` file. Using these actions, the agent is expected to either press or release these actions themselves in the Godot engine. Godot contains a global Input class, with two functions that can manipulate the state of certain actions, namely the `action_press()` and `action_release()` functions. Using these, the agent can set their action a_t for the next timestep.

Table 2: Overview of configuration settings

Setting	Type	Default	Description
`custom_agent`	Boolean	true	Enables the use of a custom agent. Uses a the default agent class when false.
`running_scene`	String	`""`	Sets the scene being ran by Godot. Uses the main scene set in Godot when left empty.
`inputs`	Array[String]	`[]`	Names of the inputs registered in Godot that the agent should have access to. A list of all options can be found in the Godot editor, $Project \rightarrow ProjectSettings \rightarrow InputMap$.
`step_duration`	Float	`0.0`	Step size given to all nodes in the scene during `process()` and `physics_process()` calls, also known as `delta`.
`env_node_paths`	Array[String]	`[]`	An array filled with the nodes that is sent as the state to the agent class.
`reward_node_path`	String	`""`	The name of the node that should be sent as the reward to the agent class.
`MCTS`	Boolean	false	Boolean as to whether the framework should initialize and use a second SceneTree used as a FM.
`MCTS_timeout`	Float	`0.0`	The time in real-time seconds that the agent is allowed to use the FM set by the `MCTS` setting before it has to make a step in the original simulation.
`random_seed`	String	`""`	The seed used for Godot's built-in RandomNumberGenerator class. This seed needs to be set in order for the FM to work as intended.

Table 3: Overview of over-writable Agent functions

Function	Description	Called During
`initialize()`	Called with the `train`, `save_data_path` and `load_data_path` parameters. Used to load in data and initialize any needed data structures.	Called once before the main gameplay loop has started.
`select_actions()`	Sets whether each action is pressed or not. This is done manually within the function by calling Godot's Input class and *not* returned.	Gets called at the start of each new simulation step.
`update()`	Used to update any internal data structures using its new variables.	Gets called each new simulation step.
`set_environment()`	Passes the environment node array as a parameter to the agent.	Gets called each new simulation step before `update()`.
`set_reward()`	Passes the reward as a parameter to the agent.	Gets called each new simulation step before `update()`.
`exit()`	A function that lets the agent finish up whatever needs to be done. Mainly used to save data after training.	Gets called when the framework closes.

4.2 Design

This section is closely tied to section 4.3 that discusses Godapt's limitations, as certain limitations were the reason behind some design choices and vice versa. This section starts out with an explanation of the current design and inner workings of Godapt. This will be followed up by an explanation of the design philosophy, along with other designs that were considered and the reason behind their failure.

The program starts out by calling a custom build of Godot. There are only a few changes within this version of Godot, with the most important one being the possibility to create multiple SceneTrees and the ability to call their `_process()` and `_physics_process()` functions from within the GDScript side of the program. This feature enables the possibility to run Godot one step at a time, along with the potential to create several different SceneTrees at the same time. After calling the custom executable of Godot, it gets fed a custom Mainloop class, called `Framework.gd`. This is where a vast majority of processes take place.

Now that Godot considers Godapt's framework as a Mainloop, it will keep calling its `_process()` function as it would for any other game instance. A diagram depicting this function and the classes it calls can be found in figure 3. After the framework initializes both an instance of the user made custom Agent class and the CustomLoop class, it enters its loop in `_process()`. This loop follows a classic approach to RL. It starts out simulating one in-game frame in the environment by calling the `one_step()` function on the CustomLoop class. This simulates both a step in `_process()` and `_physics_process()` within the game environment. After doing so, the framework collects the environment nodes, the reward node and checks if the game is over. If not, it forwards the

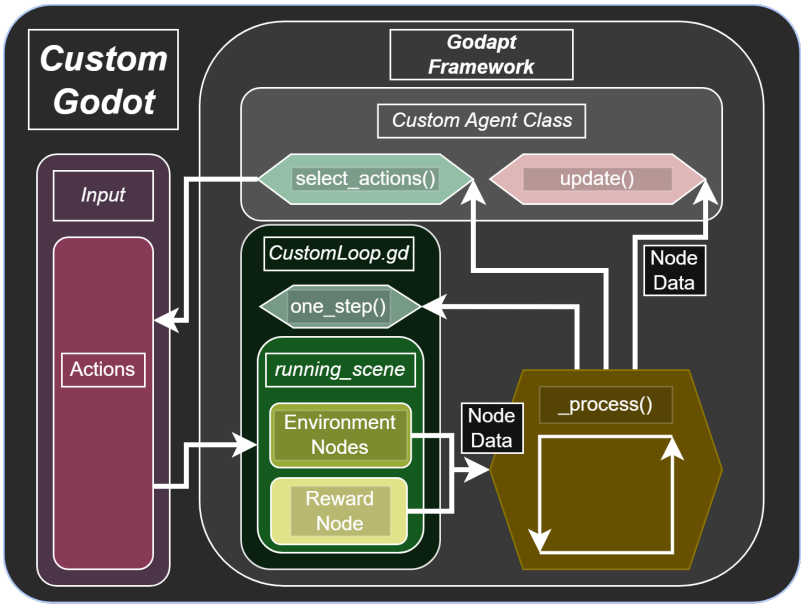


Figure 3: A simplified diagram depicting Godapt's design without the FM implementation.

environment and reward nodes toward the custom Agent, and calls its `_update()` function. After this, the agent is prompted to select its actions for the next timestep, thus completing the full RL loop.

The previous paragraph was an example of the *learning* track of Godapt. The *planning* track is different, but works in a similar manner. Please keep in mind that the planning track is currently not in a stable state, which will be discussed later on. Instead of constantly prompting the same CustomLoop object to simulate its environment, the planning track instead creates two distinct CustomLoops, each with its own scene. One of these is designated the main tree, while the other is used as a FM. In this situation, the framework switches between the two. It first simulates a single step in the main tree, after which it switches to the secondary FM tree. When this happens, a timer starts. As long as this timer is running, the framework keeps calling the FM instead of the original. During this time the agent can use planning algorithms such as MCTS or RHEA to determine its next step within the original tree. Once the timer runs out, another single step can be taken in the original SceneTree, thus completing the loop.

The way that the secondary SceneTree loads in a copy of the original tree is suboptimal. The reason as to why it is designed this way, is discussed in section 4.2.1. The primary SceneTree keeps track of all the moves it has performed up until this point, by registering them in an array. The secondary SceneTree loads in the same scene file as the original SceneTree, meaning that it is loading in the same scene, yet it starts from the very beginning. To properly make sure that the secondary tree starts out as a copy of the original environment, as is required of a FM, the `_process()` function dedicates time to simulate ever single timestep with the exact same actions that the original SceneTree has taken. This is also the reason why a random seed has to be set in the settings in order to use the FM, otherwise it would not be able to exactly recreate the same environment. The use of an internal RandomNumberGenerator object is also not allowed within the game project, as it would not fall under the same random seed as the global object. Once the

secondary tree has reached the same state as the original, it can be used as previously stated. If the secondary tree reaches an end state before the timer runs out, it will have to re-simulate back to the state of the primary tree.

4.2.1 Design Rationale

While most design choices are either trivially explained or were small enough to end up not influencing the program in any substantial way, some ended up having big effects on the outcome of this research. The following examples ended up significantly impacting the framework itself.

Why is everything done within the MainLoop's process function, and not within an internal loop?
An internal loop has been considered and tried out. This loop was called from the process function, thus decoupling the MainLoop from the rest of Godot, seeing as it would never leave the process function after entering this loop. While most parts of Godot were still working fine, this ended up having adverse side effects, such as physics interpolation no longer functioning correctly. Another major issue that this brought with it is the inability to properly render the game to a window. While rendering is not needed for a proof-of-concept, it made debugging extremely hard. These two issues combined were the reason to choose to handle things in the process function.

Why does the FM SceneTree scene need to built from scratch for every simulation?
Several different approaches of saving and loading were considered and explored through implementation. The obvious approach would be to see if it would be possible to either duplicate the original scene directly, or to save and load the scene using one of Godot's built-in features. The SceneTree itself can not be duplicated, due to its singleton attribute, though only the scene itself needs to be copied. Unfortunately, there is no built-in way to directly duplicate a full scene within Godot. To try to achieve this, two approaches were tested. The first approach consisted of saving the entire scene in the TSCN (text scene) format. However, this format does not support dynamically allocated nodes and variables, meaning it only saves the initial state of the scene and not the current state, thus making it unfit for Godapt's purposes. The second approach consisted of copying over every node into the new SceneTree. Each node had its data stored, which consisted of its node type, its place within the tree, all inspector values, any attached script and all variable values within that script. These nodes were then recreated one by one inside the new SceneTree, thus recreating the original. This approach proved successful for trying to recreate the scene itself. Two main problems arose, however. First of all, these new nodes were added into the SceneTree, meaning their `_ready()` function was called once again, leading to unexpected behavior. While turning off the `_ready()` function when adding them to a new SceneTree by making an exception within the engine-side could have been used as a solution for this, some nodes relied on the `_ready()` function to link either their signals on certain variables that were linked or set within the `_ready()` function. Though a working version of this implementation was not successfully created, the potential of it will be discussed further in section 6.

Why can the agent only be developed in GDScript?
This is because it does not matter for the research question whether the agent can be developed in GDScript or Python. It has been proven before by different projects that it is possible to interact with Godot through other scripts. One of these ways is the approach of GymGodot, which sets up a WebSocket class that listens to calls from a Python server. Another approach was taken by

GodotAIGym[4], a project similar to GymGodot. GodotAIGym allocates a block of shared memory that both Python and GDScript can access, thus setting up a way of communication between them. Setting things up exclusively using GDScript allowed Godapt to be developed faster and with a focus that lies more on manipulating the engine into allowing the creation of a FM.

4.3 Limitations

Godapt suffers from a collection of limitation. Some of these are due to the design of the software, while other can be considered bugs. These range from hard restriction to minor inconveniences, and will be separated into two degrees of severity accordingly. For a deeper explanation for some of the major limitations, refer back to section 4.2.1.

4.3.1 Major Limitations

The array tracking the moves of the primary SceneTree grows linearly larger with each timestep, meaning the longer the game keeps running, the larger this array grows.
This means that it is impossible for a game to keep running indefinitely, and weaker machines will quickly run out of available memory in their heap. This will eventually lead to the program crashing.

The game needs a set random seed in order to use the FM.
In order to recreate the scene from scratch, the secondary tree needs to replicate every move made so far. Because of this, it can not use a different seed each time, otherwise the result would vary and it would not properly simulate the primary SceneTree. This also leads to the unfortunate consequence that agents will never have to keep a different random state in mind, but instead will always play the same game.

The game can not use an instance of the RandomNumberGenerator when enabling the FM.
As discussed earlier, the secondary tree needs to replay the game from scratch to replicate the original game state. Many games, however, make use of the class RandomNumberGenerator instead of the global random functions. This would exclude them from being played using Godapt.

The game can not use UI or other mouse position related events.
Godapt exclusively makes use of the Input class in Godot. This means that keeping track of mouse position is currently impossible. This limitation excludes games that rely on mouse position clicks, such as most Real Time Strategy (RTS) games or top down Role Playing Games (RPG).

The game needs to reset the scene themselves whenever it ends.
There is currently no support for resetting scenes through Godapt. This is because the framework was developed with the philosophy that a game would not have to be edited in order to be compatible with Godapt. In reality, this leads to many different issue and limitations, though they do not impact the research itself. This issue could be fixed by finding a balance between leaving the game untouched and editing/adding nodes.

[4] https://github.com/lupoglaz/GodotAIGym

4.3.2 Minor Limitations

The custom agent needs a reboot of the Godot editor.
This project has tried to keep the already developed game and the framework separate. A newly created custom Agent class, however, needs to be registered within the project, and the only way to do so currently is to boot up the specified game within the Godot editor. This could possibly be solved with a script on the custom engine side, but this was outside the scope of this research.

The loop being in `Framework.gd` limits the use of Godapt for uncommon purposes.
Currently, Godapt is always stuck in the same loop, with the only user-made interaction being inside the Agent class. There are various reasons why the user would want to control and customize this loop for themselves, for example: having several agents at the same time. This is currently not possible without editing Godapt's files yourself.

The step size (delta) is always set.
This was done for the ease of development, as it made creating example agents significantly easier, and had no adverse effects on the research itself. This leads to the agent never learning to keep delta in mind, which sometimes a wanted feature when developing an agent.

All code is limited to GDScript.
Python is the industry standard for AI programming, due to its vast amount of libraries available for data manipulation. GDScript does not have access to any libraries, causing programmers to have to program complex algorithms and data structures from scratch. While technically the same things are possible in both programming languages, doing so in GDScript is oftentimes implausible.

Godapt exists with a plethora of errors and memory leaks.
Due to our design approach, we are instantiating several singelton objects. Godot is not designed with this action taken into consideration, leading to unintended behavior and memory leaks when Godot tries to clean up after running. No adverse side effects have yet to be discovered during runtime, but not cleaning up memory is generally considered poor practice.

5 Experiment

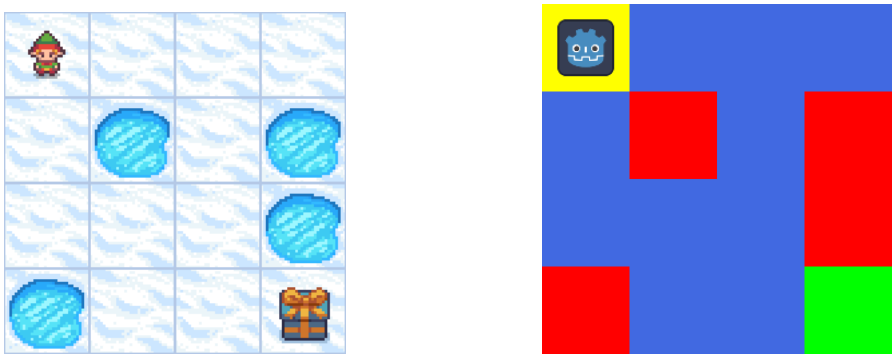
In order to make sure Godapt worked as intended, test projects were set up. Not all results will be shown, as a working random agent is trivial when RL agents in the same environment have been shown to perform as intended. Therefore, the two most significant projects will be shown. The only proper experiment that has been sit up discussed thoroughly in section 5.1. This is due to the fact that this is the only experiment with quantifiable data. Most features, such as letting agents interact with several different games, are proven empirically and do not have a separate experiment set up. The planning track is not discussed here, seeing as whether the current implementation works, does not change the outcome of this research.

5.1 Frozen Lake (Learning)

To test the possibility of creating learning agents within Godapt, a recreation of OpenAI's Gymnasium Frozen Lake environment was created within Godot, which can be seen in figure 4. Apart

from slight differences, such as the Godot version resetting automatically and making use of the Input class for actions, they function identically.

For this experiment Q-learning was chosen in combination with the ϵ-greedy policy as the RL agent's algorithm, due to the following reasons. First of all, its simplicity not only makes it easy to implement, it also makes it easier to check if all features, such as the learning loop or reward system, work as intended. Secondly, Q-learning produces repeatable and interpretable data, making correctness verification easier by looking at the reward per timestep or the Q-table itself. Thirdly, the algorithm can be easily implemented without the use of Python libraries, leading to significantly faster development time in GDScript.



(a) The standard 4x4 Frozen Lake environment. Image from the official website.

(b) Godot recreation of the official 4x4 Frozen Lake environment.

Figure 4: A visual comparison between the original Python Frozen Lake environment and Godapt's adaptation. In Godot, yellow indicates the start, green indicates the present (goal), red indicates a hole (reset) and blue indicates walkable tiles.

The hyperparameters for the following experiments are: $\epsilon = 0.1$ and $\gamma = 0.9$. Both the Python and Godapt implementations were tested using learning rates α of 0.5 and 0.1. These values are commonly recognized as effective defaults for Q-learning in most environments. Seeing as the main focus of these experiments is to examine if an agent in Godapt learns correctly and similarly to an agent in Python, attention to these parameters will be limited. The reason for having two varying learning rates, is to see if both agents react similarly to a change of parameters. Each experiment was ran 100 times, with each attempt lasting 3000 episodes. Due to the large amount of episodes and the sporadic nature of the recorded rewards, a rather high window size of 200 has been chosen for the rolling window calculation smoothing.

To make the comparison as fair as possible, both the Python and Godapt agent were programmed with the same functions called in the same order. All code from the GDScript agent was copied to Python and adjusted when needed. The only major change in behavior is that the Godapt agent has to call the Input class to register an action, while the Python agent registers its chosen action as a return variable.

The results shown in figure 5 indicate no significant difference between the learning capabilities of an agent developed within Python compared to Godapt. The slight differences between the two can be attributed to the inherent randomness in RL and the stochastic nature of the ϵ-greedy policy.

These findings suggest that RL agents developed in Godapt are capable of learning correctly and performing comparably to those created in more established environments.

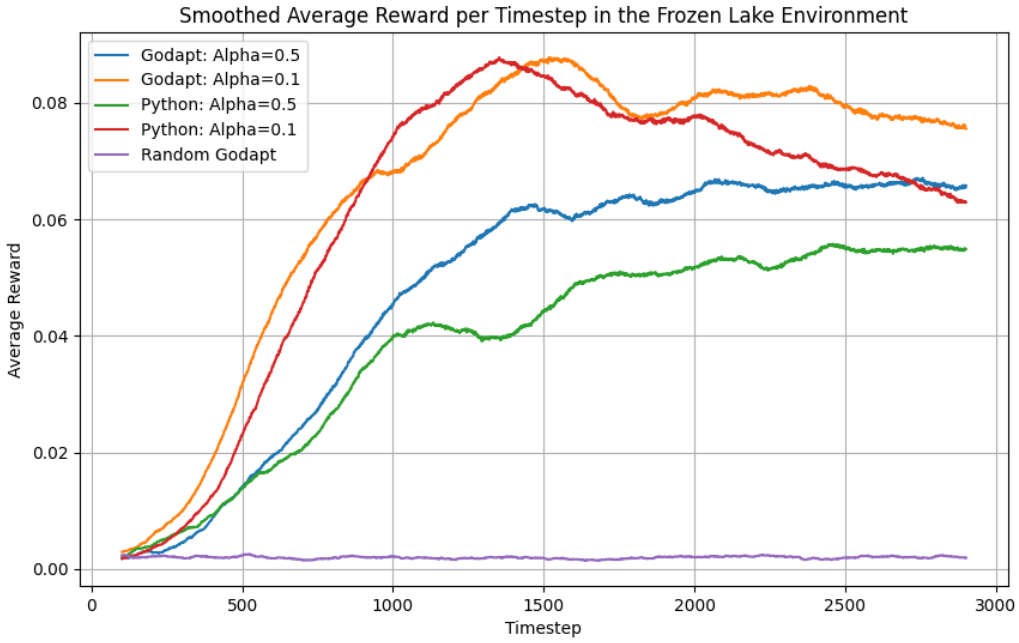


Figure 5: A comparison of Q-learning agents in both Python and Godapt in the Frozen Lake environment, each with a learning rate of 0.1 or 0.5, accompanied with the results of a random agent. All results have been smoothed with a window of 200.

6 Conclusions

This research has introduced a framework called Godapt, implemented within Godot, that lets its users develop and train AI agents that utilize the engine's games as environments. Using Godapt we have empirically shown that it is possible to create Reinforcement Learning agents that train on arbitrary Godot scenes directly within the engine.

There was less success in the development of planning agents, however. While we have attempted many different ways to get a FM to work within Godot, the engine simply has not been implemented with its support in mind. We have managed to find a workaround, though it imposes severe limitations. Because of this, we conclude that a FM is currently *not* possible in the way we imagined it.

6.1 Further Research

In order to prove the research question, the creation of a proper FM must be possible. To achieve this, a feature must be developed that lets the framework duplicate existing scenes. As of now, Godot 4 will not be getting a dynamic scene duplication feature. A promising alternative was an

approach considered and implemented within this research, namely the copying of all node data into a newly created scene. While we were not able to create such a feature successfully, we believe that there is potential in this approach, either through implementing it engine-side or by doing it within the framework itself. Though many edge-cases must most likely be considered, implementing such a feature would allow the creation of several FMs and thus prove the research question.

Though the FM is the main point of attention, many other angles and aspects of Godapt can use improvements. As discussed in section 4.3, there are currently a collection of limitations plaguing Godapt. Most of the major limitations are tied to the current implementation of the FM feature. Many of the minor limitations can be fixed by adding or changing certain aspects of Godapt. For example: Godapt could be improved by adding cursor position support, letting agents keep track of the mouse's screen position. A major overhaul could also allow the creation of agents within other programming languages, as has been proven by projects such as GymGodot.

New features tying Godapt to GVGAI could possibly be added. For example: a script that lets people convert their old GVGAI agents into Godapt compatible agents. While GVGAI has stopped organizing its yearly competitions, similar competitions could be organized within Godapt. While currently not as robust as GVGAI, Godapt could organize such competitions even in its current state, seeing as all GVGAI competitions solely took place in its learning track.

References

[AFF+22] Joshua Albrecht, Abraham J Fetterman, Bryden Fogelman, Ellie Kitanidis, Bartosz Wróblewski, Nicole Seo, Michael Rosenthal, Maksis Knutins, Zachary Polizzi, James B Simon, and Kanjun Qiu. Avalon: A benchmark for RL generalization using procedurally generated worlds. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022.

[BCP+16] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[BNVB13] M.G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[CHhH02] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[CWU+08] G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 2008.

[ELL+13] M. Ebner, J. Levine, S.M. Lucas, T. Schaul, T. Thompson, and J. Togelius. Towards a video game description language. *Dagstuhl Follow-Ups*, 6, 2013.

[GRLPL17] C. Guerrero-Romero, A. Louis, and D. Perez-Liebana. Beyond playing to win: Diversifying heuristics for gvgai. *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 118–125, 2017.

[JHHB16] M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. The malmo platform for artificial intelligence experimentation. *IJCAI*, 2016.

[LCE+13] J. Levine, C. Bates Congdon, M. Ebner, G. Kendall, S. M. Lucas, R. Miikkulainen, T. Schaul, and T. Thompson. General video game playing. *Dagstuhl Follow-Ups, Volume 6*, 2013.

[PLLG+19] D. Perez-Liebana, S.M. Lucas, R.D. Gaina, J. Togelius, A. Khalifa, and J. Liu. *General Video Game Artificial Intelligence*, volume 3. Morgan & Claypool Publishers, 2019. https://gaigresearch.github.io/gvgaibook/.

[SB20] R.S. Sutton and A.G. Barto. *Reinforcement learning: An introduction (2nd ed.)*. MIT Press, 2020.

[Sch99] S. Schaffer. *"Enlightened Automata", in Clark et al. (Eds), The Sciences in Enlightened Europe*. The University of Chicago Press, 1999.

[SLPL+18] C.F. Sironi, J. Liu, D. Perez-Liebana, R.D. Gaina, I. Bravi, S.M. Lucas, and M.H. Winands. Self-adaptive mcts for general video game playing. In *International Conference on the Applications of Evolutionary Computation*, pages 358–375. Springer, 2018.

[TBT+18] R. Torrado, P. Bontrager, J. Togelius, J. Liu, and D. Perez-Liebana. Deep reinforcement learning for general video game ai. *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2018.