

**A Research Framework for Asynchronous Adversarial Multi-Player Games with  
Human Player GUI and AI Gym**

by

Cody Roberts

A thesis submitted to the Graduate Faculty of  
Auburn University  
in partial fulfillment of the  
requirements for the Degree of  
Master of Science

Auburn, Alabama  
August 5, 2023

Keywords: adversarial games, asynchronous, multi-player, AI Gyms

Copyright 2023 by Cody Roberts

Approved by

Daniel Tauritz, Chair, Associate Professor of Computer Science and Software Engineering  
Samuel Mulder, Associate Research Professor of Computer Science and Software Engineering  
Davide Guzzetti, Assistant Professor of Aerospace Engineering  
Akhil Rao, Assistant Professor of Economics, Middlebury College

## Abstract

The primary contribution of this thesis is to describe the design and development of a research framework to support complex, real-time asynchronous multi-agent simulations with both homogeneous as well as heterogeneous human and artificial intelligence (AI) agents. This framework provides researchers with a platform to model adversarial games and benchmark AI algorithms and policies. It provides a flexible, reusable client/server architecture that supports a wide variety of games for use as (learning) environments.

The secondary contribution of this thesis is to describe the design and development of an example use-case of the research framework for satellite constellations named Satellite Tycoon (Sat-Tycoon). The provided React client provides researchers, educators, and aerospace enthusiasts with a way to play the provided Sat-Tycoon game from their internet browser without the need for downloading or installing. The Sat-Tycoon game itself is a satellite constellation simulation rich with complexity, ideal for use as a reinforcement learning environment.

This thesis will cover the design decisions made for these contributions, and the trade-offs involved in those decisions. Such decisions include using multiple networked gyms instead of a single multi-agent gym, using the OpenAI gym standard instead of PettingZoo, building a server authoritative architecture over a peer-to-peer architecture, and using a web framework instead of a game engine.

## Acknowledgments

The work described in this thesis received funding from the Air Force Research Laboratory (AFRL)<sup>1</sup> under contract FA9453-20-1-0008 as well as from the Auburn Cyber Research Center (ACRC)<sup>2</sup>, without their generous support this work would not have been possible.

I would like to acknowledge my committee chair and advisor Dr. Daniel Tauritz, without his guidance and relentless support I would not have been able to reach this major milestone in my life. I appreciate his unwavering encouragement and optimism, which kept me going through the most difficult challenges. I would also like to express my deepest gratitude to my committee members: Dr. Davide Guzzetti, Dr. Samuel Mulder, and Dr. Akhil Rao for their unique perspectives, incredible wisdom, and invaluable feedback. I would also like to thank my teammates: Rehman Qureshi, Jay Patel, Manuel Indaco, Lucy Bone, and Emily Kimbrell for their friendship, hard work, and contributions to the project. I would like to give special thanks to my good friend Deacon Seals, whose guidance, knowledge, friendship and support proved to be indispensable. From my bachelor's degree to my master's, Deacon has been there to help and support me late into the night and I couldn't have done it without him. I want to thank my parents Aaron and Bernetta Roberts for always emphasizing the importance of education and supporting me in my academic endeavors, and my brother Keith Roberts for always being there for me when I needed him. Finally, I want to thank my best friend and partner in all things Meredith Brown for all of her love and support, and having the patience to see me through this journey.

---

<sup>1</sup><https://www.afrl.af.mil/>

<sup>2</sup><https://cyber.auburn.edu>

## Table of Contents

Abstract . . . . .	ii
Acknowledgments . . . . .	iii
List of Figures . . . . .	ix
List of Tables . . . . .	x
1 Introduction . . . . .	1
1.1 Reinforcement Learning . . . . .	2
1.2 AI Gyms . . . . .	2
1.3 OpenAI Gym . . . . .	3
1.3.1 Step Function . . . . .	3
1.3.2 Reset Function . . . . .	4
1.3.3 Spaces . . . . .	4
1.3.4 Action Space . . . . .	4
1.3.5 Observation Space . . . . .	4
1.4 DotA 2 and StarCraft II . . . . .	5
1.4.1 The Sat-Arena Gym for Sat-Tycoon . . . . .	6
1.5 Sat-Tycoon . . . . .	9
1.5.1 Simulation of Time . . . . .	9
1.5.2 Making Actions . . . . .	10
1.5.3 Direct Player Interaction . . . . .	10

1.5.4	Information Framing . . . . .	10
1.5.5	Ending the Game . . . . .	10
1.6	Contributions . . . . .	10
1.7	Outline . . . . .	11
2	Related Work . . . . .	12
2.1	OpenSpiel . . . . .	12
2.2	PettingZoo . . . . .	12
2.2.1	Agent Environment Cycle . . . . .	13
2.3	Dopamine . . . . .	13
2.4	Related Aerospace Gyms . . . . .	14
2.4.1	Deep Replacement: Reinforcement learning based constellation management and autonomous replacement . . . . .	14
2.4.2	StarCraft II . . . . .	14
2.5	Multi-Agent Reinforcement Learning . . . . .	15
2.6	Multi-Agent Reinforcement Learning Gyms . . . . .	15
2.6.1	TanksWorld . . . . .	15
2.6.2	Battlesnake Challenge . . . . .	16
2.7	Competitive Multi-Agent Environments . . . . .	17
2.7.1	NeuralMMO . . . . .	17
2.7.2	Online RPG For Reinforcement Learning . . . . .	18
3	Client/Server Architecture and Model/View Separation . . . . .	20
3.1	Clients . . . . .	20
3.2	Server . . . . .	20
3.2.1	Lobbies . . . . .	20
3.2.2	Users . . . . .	21

3.2.3	Multiprocessing . . . . .	22
3.2.4	Game Logic . . . . .	23
3.2.5	Ending the Game . . . . .	24
3.3	Multiprocessing . . . . .	24
3.4	Server Authoritative Architecture . . . . .	24
4	The React Client for Human Players . . . . .	28
4.1	Player Login . . . . .	29
4.1.1	Zero Turn . . . . .	29
4.2	Components . . . . .	31
4.2.1	Footer . . . . .	31
4.2.2	Multi-Panel . . . . .	32
4.2.3	Earth Panel . . . . .	32
4.2.4	Space Panel . . . . .	32
4.2.5	Social Panel . . . . .	35
4.2.6	Analytic Panel . . . . .	35
4.2.7	Scoreboard . . . . .	36
4.3	Redux State . . . . .	36
4.4	Networking . . . . .	37
4.4.1	API . . . . .	38
5	The Sat-Arena OpenAI Gym Client . . . . .	39
5.1	Simplified Environment . . . . .	39
5.2	Step . . . . .	40
5.3	Reset . . . . .	40
5.4	Timesteps . . . . .	41
5.5	Networking . . . . .	41

5.5.1	The Connection Class . . . . .	41
5.5.2	Connecting . . . . .	42
5.6	Multiprocessing . . . . .	42
5.6.1	Action Queue . . . . .	42
5.6.2	Response Queue . . . . .	43
5.6.3	Connection State Dictionary . . . . .	43
6	Sat-Tycoon as a Research Framework . . . . .	44
6.1	Supported Games . . . . .	44
6.1.1	Markov Decision Process . . . . .	44
6.1.2	Extensive-Form Games . . . . .	45
6.2	Independent Gym Environments in Multi-Agent Learning . . . . .	45
6.3	Client Agnostic Design . . . . .	45
6.4	Possibilities . . . . .	46
6.5	Results . . . . .	46
7	Conclusion . . . . .	48
7.1	Limitations . . . . .	48
7.1.1	Reinforcement Learning . . . . .	48
7.1.2	Design . . . . .	49
7.1.3	Hardware . . . . .	49
7.1.4	Networking . . . . .	49
7.2	Future Work . . . . .	49
7.2.1	Converting React Client to Functional Components . . . . .	50
7.2.2	More Complex Gym Environments . . . . .	50
7.2.3	Mini-games and Challenges for RL Agents . . . . .	50
7.2.4	Visualizer Client . . . . .	50

7.2.5	Server-side Gym Environments . . . . .	51
7.2.6	Persistent Data . . . . .	51
References	. . . . .	52
Appendices	. . . . .	55
A	Changing the Environment . . . . .	56
A.1	Changing the Game Logic . . . . .	56
A.2	Changing the API . . . . .	57
A.3	Other files . . . . .	58
A.4	Developing clients . . . . .	58
B	Getting Started . . . . .	59
B.1	Setting up Linux . . . . .	59
B.1.1	Setting up Linux using WSL2 in Windows 10/11 . . . . .	59
B.1.2	Setting up a dedicated Linux install . . . . .	60
B.2	Using Git and GitHub . . . . .	60
B.3	Installing the Backend . . . . .	61
B.4	Recommended Development Environment . . . . .	61
B.5	Running the Server . . . . .	62
B.6	Installing the React Client . . . . .	62
B.7	Installing the Gym Client . . . . .	62
B.8	Hosting the Server . . . . .	63
B.9	Hosting the React Client . . . . .	63



## List of Figures

1.1	An agent sends actions to an environment which returns the next state and reward.	3
1.2	Two alternatives for implementing multi-agent gyms in the framework . . . . .	8
3.1	The server organizes players into lobbies. . . . .	21
3.2	The exchange of data between the server and a gym client, and the internal data flow for each. . . . .	27
4.1	The initial title screen for the React player client. . . . .	29
4.2	The login page allows players to choose a name and lobby. . . . .	30
4.3	The lobby page shows which players have joined the game, and lets a player signal they are ready to play. . . . .	30
4.4	The zero turn provides a tutorial experience for players who may be new to the game. . . . .	31
4.5	The footer provides a clear picture of a player’s resources during gameplay. . .	32
4.6	The multipanel serves to help players access multiple areas of the game from one location. . . . .	33
4.7	The Earth panel provides access to features such as the world map and ground-station placement. . . . .	34
4.8	The Space panel provides access to features such as the Space Deck and launching satellites. . . . .	34
4.9	The Social panel provides a place for players to interact with each other directly.	35
4.10	The Analytic panel gives players important statistical data about the game. . . .	36
4.11	The scoreboard displays how well players fared against their competitors after a game has ended. . . . .	37
6.1	Output from a successful game between two agents using the A2C model in Stable Baselines 3. . . . .	47

## List of Tables

2.1	Table comparing different RL & MARL frameworks . . . . .	13
2.2	Table comparing different competitive multi-agent environments. . . . .	18

## Chapter 1

### Introduction

Many real-world situations are naturally modeled as asynchronous adversarial multi-player games, from economic competitions to warfare. However, there is a lack of research frameworks capable of modeling such games with support for both human players via GUI clients and AI players via standardized AI gym interfaces. This thesis describes the creation of such a framework for the specific scenario of economic competition in satellite constellations, both the general components and the scenario specific ones.

Reinforcement Learning (RL) is one of the primary types of machine learning (ML), a subset of Artificial Intelligence (AI) [16]. It is of critical importance to the development of RL to have high-quality standardized benchmark environments to perform rigorous comparisons of different RL algorithms, often referred to as AI gyms. The primary contribution of this thesis is to describe the development of a framework to support complex, state-of-the-art real-time asynchronous multi-agent simulations with human involvement.

Satellite Tycoon, further referred to as Sat-Tycoon for short, is a multiplayer satellite constellation strategy-simulation game reminiscent of Tycoon Games (e.g., Zoo Tycoon, Jurassic World Evolution, Railroad Tycoon, Plague Inc., and RollerCoaster Tycoon). These games tend to border on economic simulations where the goal is to develop a strategy to build the most profitable business. Players in Sat-Tycoon compete to see who can build the most profitable satellite constellation to provide internet service to different regions of the world.

Sat-Tycoon presents AI with a complex, adversarial, multi-agent economic competition. Using a custom framework it provides a real-time asynchronous competitive environment in which to benchmark against both other AI agents as well as human players. This type of

framework should be of particular interest to the RL community. The author has done an admittedly non-comprehensive review of several similar environments and found this framework to be novel in supporting asynchronous real-time training and competition over the internet. This review can be found in Chapter 2.

This framework provides a general framework for research supporting custom built clients using WebSockets and a JSON API. It also provides two premade clients, one for humans using JavaScript and React, and one for RL agents using Python and OpenAI Gym. The benefits of this framework for researchers are covered in more detail in Chapter 6.

## 1.1 Reinforcement Learning

Reinforcement Learning (RL) is a method to train AI agents through trial-and-error, using a pre-constructed environment. This environment is primarily exposed to the agent as a set of observations, actions, and rewards. Much work has been done in the field of RL, with many rich, challenging environments being developed for agents to learn in. Many of these environments have been created using OpenAI's Gym [3] API. OpenAI has set the standard for RL environments with their Gym framework, which is one of the reasons Sat-Tycoon's AI learning environment was originally built using OpenAI Gym. As of the writing of this Thesis, all maintenance and development of OpenAI Gym has ceased and the development team has forked the project to be continued as Gymnasium<sup>1</sup>. However, the provided AI gym client originally used OpenAI Gym and was converted to Gymnasium during the writing of this thesis. This thesis will discuss the gym version of the provided AI client.

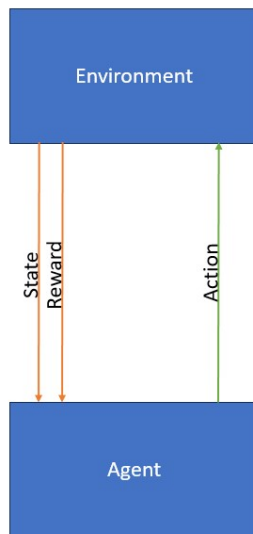
## 1.2 AI Gyms

AI Gyms are environments designed and developed to provide an easy API for training RL agents by playing games. Their most prominent use among the RL community is to benchmark and compare RL algorithms. One of the leading standards in AI gyms is the OpenAI Gym API.

---

<sup>1</sup><https://gymnasium.farama.org/>

Figure 1.1: An agent sends actions to an environment which returns the next state and reward.



### 1.3 OpenAI Gym

OpenAI Gym is a framework developed for building environments in which to train RL agents. OpenAI’s gym API has become one of the leading standards in the RL community for training RL algorithms. It utilizes an episodic structure that seeks to maximize total reward per episode. OpenAI is well known for their work in developing agents to play the game DotA 2, a real-time asynchronous game that faces many challenges similar to Sat-Tycoon. They have also done important work in many other areas of AI, including their large language models such as GPT-3. Sat-Tycoon includes an OpenAI Gym environment called Sat-Arena to allow agents to train on the Sat-Tycoon game. The Sat-Arena environment is covered in more detail in Section 1.4.1.

Figure 1.1 visualizes a simple example of an OpenAI Gym.

#### 1.3.1 Step Function

The step function is the main function of an AI gym environment. It receives the action an agent intends to take, and the current observed state. It returns four variables; *state*, *reward*, *info*, and *done*. The state is the next state, calculated using the action provided. The reward is calculated based on a reward function defined by the gym creator, and is used to give the

agent feedback on what is or isn't good. The info variable is used to communicate any extra information such as debugging data, or sometimes extra information for libraries such as stable baselines to use. The done variable is used to determine when the game has ended and a new game should be started.

### 1.3.2 Reset Function

The reset function serves to clean the slate after an episode and prepare the environment to run new experiments. It should clear out all of the variables, and return the environment to its initial state.

### 1.3.3 Spaces

Spaces are data structures used by Gyms to define the format for actions and observations. This dictates what actions must look like for agents and also defines what observations will look like. A space can be formatted in many different ways depending on the environment's needs. The space types supported by OpenAI Gym are Box, Dict, Discrete, Graph, MultiBinary, MultiDiscrete, Sequence, Text, and Tuple. In Section 5.1, Sat-Tycoon's use of these types is detailed.

### 1.3.4 Action Space

Action spaces are used by AI Gym environments to represent the set of all actions that can be taken by an agent, as well as legal ranges for those actions. For example, in Sat-Tycoon the action space dictates that the inclination for a satellite launch must be between 0 and 180 degrees. Any action an agent takes must be selected from the corresponding action space for the environment and fall within the provided valid parameters.

### 1.3.5 Observation Space

Observation spaces are used by AI Gym environments to represent what the environment the agent is exploring looks like. They should contain all of the information an agent would need to learn and play a game. For example, an observation space may contain spatial and positional

data or even quantitative data. In Sat-Arena, these are matrices describing a wide variety of things ranging from global population data on Earth to the locations of satellites in space for each player.

#### 1.4 DotA 2 and StarCraft II

DotA 2<sup>2</sup> is a Multiplayer Online Battle Arena (MOBA) developed by Valve<sup>3</sup> in which ten human players compete against each other in two teams of five. Each player is in control of a single hero, and must manage that hero's items and abilities, level up the hero and select new abilities each level, as well as work together with teammates to complete objectives. DotA 2 is a highly complex environment, involving a roster of more than one hundred selectable heroes, a partially observable state, and dozens of other units and buildings. OpenAI<sup>4</sup> developed their RL agent OpenAI Five [2] to play DotA 2. OpenAI Five trained by playing 10,000 years of DotA 2 matches against itself. It handles the complexity of DotA 2 by discretizing the action space, and setting a few limitations. It only supports 17 of the total of 117 heroes in DotA 2, and does not support any items which require a player to temporarily control multiple units at the same time. In April 2019, OpenAI's agent OpenAI Five beat the DotA 2 world champions<sup>5</sup>.

While OpenAI developed OpenAI Five to play DotA 2, DeepMind developed AlphaStar<sup>6</sup>, the first AI to beat professional StarCraft II players. StarCraft II<sup>7</sup> is a real time strategy game developed by Blizzard Entertainment. Unlike DotA 2, in StarCraft II the player is in control of an entire army instead of a single hero. The player must manage resources for production of units, construction of buildings, and completion of goals like defeating enemy players in battle. A game generally consists of two to eight players of varying team sizes. StarCraft II is a partial information game, only revealing to players their own information, what their units can see, and parts of the map they have explored. Opposing player information is hidden, and unexplored portions of the map are covered in a "fog of war", preventing the player from

---

<sup>2</sup><https://www.dota2.com/home>

<sup>3</sup><https://www.valvesoftware.com/en>

<sup>4</sup><https://openai.com/>

<sup>5</sup><https://openai.com/research/openai-five-defeats-dota-2-world-champions>

<sup>6</sup><https://www.deepmind.com/blog/alphastar-mastering-the-real-time-strategy-game-starcraft-ii>

<sup>7</sup><https://starcraft2.blizzard.com/en-us/>

seeing those areas. DeepMind released the StarCraft II Learning Environment (SC2LE) [19] for RL agents to play StarCraft II. SC2LE consists of three parts: a Linux binary for StarCraft II, an API for communicating with the StarCraft II game, and the PySC2 environment. PySC2 utilizes StarCraft II's built-in map editor to create several simpler mini games for agents to tackle before attempting to learn to play the whole game. Sat-Tycoon is similar in providing both an API for communication with the game as well as a gym environment for running RL agents. The Sat-Tycoon API provides networking functionality for communication with the game server, and just like the SC2LE API it provides functionality for starting a game, sending actions, and receiving observations. It does not currently provide functionality for reviewing replays, but implementation of this functionality would not be difficult in future work.

Sat-Tycoon shares many challenges with DotA 2 and StarCraft II. Like these games, Sat-Tycoon has a partially observed state, as players cannot see the budget, revenue, number of owned satellites, or researched technology of other players. Sat-Tycoon also has a highly dimensional observation space, containing multiple grids representing maps of global data such as population, customers, data-rate, coverage, and even satellites in space. In addition to these challenges, Sat-Tycoon is also a real-time asynchronous game. Handling the step function can be challenging in real-time games, and in DotA 2, OpenAI Five conveniently addresses this by tying the step function to the frame-rate of the game. Since the Sat-Tycoon game does not have a frame-rate the step function cannot be tied to it. Instead, Sat-Arena's step function behaves in a more real-time manner, with each step not being tied to a particular environmental factor. Instead, agents may make steps as quickly as they are capable, and at the end of each month agents receive environment updates. This results in a delayed reward system, and may not uphold the Markov Property, which is covered in more detail in Subsection 6.1.1.

#### 1.4.1 The Sat-Arena Gym for Sat-Tycoon

Sat-Arena is an OpenAI Gym developed specifically for use with the framework. Written in Python, the gym offers a simplified environment to showcase the framework's ability to support RL gyms and AI. When designing a gym for use with the framework, there are two immediate options. The first option is a more traditional multi-agent environment in which several agents



play in the same gym environment, as illustrated in Figure 1.2b. Using this method, a single gym client would connect to the server to allow multiple agents to play using a single gym. Because the server treats each individual WebSocket connection as a single player, it does not currently support a gym environment where multiple agents play as different players in one gym. However, it may be possible to train multiple agents to work together to manage a single player's actions using such a gym design. This style of gym client was not chosen for Sat-Tycoon because it is not ideal for a real-time, asynchronous game. Due to the asynchronous nature, agents sharing an environment may find themselves waiting for each other. In this case if there are three agents, the first agent to finish its step function will not see the resulting state and reward from its action until the next two agents also finish making their steps. The alternative is that agents take sequential turns, which is not faithful to Sat-Tycoon's real-time asynchronous nature.

Sat-Arena solves this problem by being a single agent environment designed to network with other single agent environments, as shown in Figure 1.2a. This allows the agents to take their steps completely independently, and play at their own pace asynchronously. The agent makes their action during the step phase, and waits to hear back from the server with their corresponding result. The client receives and accrues responses from the server in the response queue until the response to the agent's action is received. Each action contains a parameter indicating which player the action is associated with, so that the listener can loop through the response queue until it finds the response to the agent's last action. In this way, the agent gets all of the environment updates since their last action.

Networking multiple single agent environments also more naturally facilitates scalability than a single multi-agent environment, as each agent can be in its own environment on its own machine, and does not have to share system resources. Alternatively, an environment could achieve more scalability by utilizing multiple cores of a single machine.

However, it is not a perfect solution and there are some drawbacks that may need to be addressed. For example, if an opponent were to send updates at a fast enough rate while the agent was looping through responses, they could lock out or delay the agent from receiving its

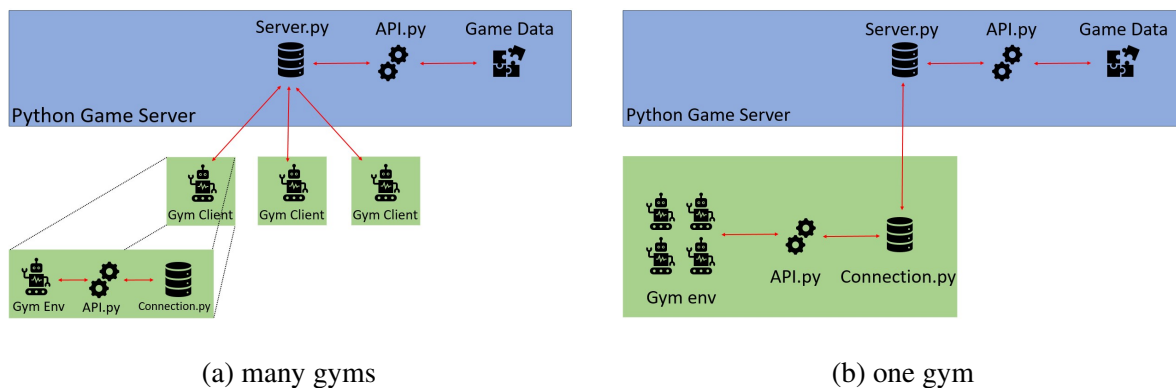


Figure 1.2: Two alternatives for implementing multi-agent gyms in the framework

own response. This wouldn't be a problem in the first design, where agents share an environment, as no agent can continue making actions until all agents have acted or agents are playing synchronously in a turn-based system. The environment is also simplified and designed specifically to work with the more rudimentary algorithms provided by the Stable Baselines library, and a more complex gym environment may need to be developed to support more advanced algorithms.

### Stable Baselines 3

Sat-Arena does not implement or provide any implementations of RL algorithms, but has been tested using the Stable Baselines 3 library [14]. Stable Baselines 3 provides a set of commonly used model-free, single-agent algorithms that have been thoroughly tested and benchmarked. Sat-Arena has been tested specifically with the Advantage Actor Critic (A2C) [11] algorithm, but there may be other suitable algorithms for Sat-Tycoon as well both within the Stable Baselines 3 library and outside of it. Sat-Arena behaves the same as any other standard OpenAI gym environment and can be used as such, allowing researchers to write and develop their own custom algorithms and agents to train in it.

### Scalability

The framework has no hard coded limit on the number of connections the server can receive at once. Complementing this, the Sat-Tycoon game itself has no maximum player limit. The advantage of this is that large scale games are inherently supported at the software level by

both the game logic and the server; however, limitations may still be encountered depending on the hardware the server is hosted on. In addition, the game mechanics may begin to break down as the number of players is increased. While Sat-Tycoon has no strictly enforced player limit, certain game elements like orbital planes on the space deck will quickly see crowding with too many players. The provided gym client is also not without scaling limitations as the method used to listen for an agent's state updates is not ideal for very large scale games. This is because the agent will listen to environment updates until it receives one that is indicated as being a result of its own action. This means that if the agent receives enough environment updates quickly enough, it may be locked out from receiving the result of its last action. This effectively prevents the agent from finishing its step function, and prevents the agent from continuing to play.

## 1.5 Sat-Tycoon

Sat-Tycoon is a competitive multi-player economic simulation played asynchronously in real-time. The goal of the game is for players to build the most profitable satellite internet service by constructing the best satellite constellation. The game runs on a simulated clock and players can make as many actions as they like, as often as they like. Players receive updates to their customer and revenue values every in-game month, representing their overall performance in the game.

### 1.5.1 Simulation of Time

The game takes place over a set amount of time measured in months and years, during which players can make any number of actions. This runs on an iterative clock that represents one second as two days. Months are normalized to thirty days for consistency, and to allow the game to be better used by any possible future local gym environments that may need a clear step function. Updates to the game state such as revenue and customers rely on the clock, arriving monthly.

### 1.5.2 Making Actions

Players have a number of actions available to them in Sat-Tycoon, all with the purpose of building a satellite constellation to provide internet service to customers. Actions can be organized into two major categories: Earth and Space. As the name implies, Earth actions are those related to more terrestrial pursuits such as building ground stations and setting the internet price customers need to pay for service. Space actions, in contrast, are related to building and launching satellites.

### 1.5.3 Direct Player Interaction

In every multiplayer game, player interaction is a vital part of the experience. Sat-Tycoon is no different, and provides player interaction via the social panel. Players can chat while playing using the chat box, or see which players are currently online and playing.

### 1.5.4 Information Framing

Sat-Tycoon is a partial information game. Players can see their own funds, internet price, ground stations, technology research and satellites. However, they can only see the ground stations and launched satellites of opponents, other opponent resources are hidden from them.

### 1.5.5 Ending the Game

When the predetermined number of years has passed, the game will end and scores will be sent to players. The scores tracked and provided at the end of the game are total number of customers and revenue gained.

## 1.6 Contributions

This thesis contributes a general framework for the modeling of real-time, asynchronous adversarial games, and a demonstration of that framework in use. The framework supports adversarial modeling for games such as economic simulations, attacker/defender games, and more using a real-time, client agnostic server authoritative architecture. This architecture makes the

framework perfect for real-time asynchronous multi-agent RL, as it allows for users to develop their own clients and thus their own gym environments. The framework is demonstrated using the Satellite Tycoon [6] game, which models economic competition from the perspective of satellite constellation management. While Sat-Tycoon simulates economic competition, it can be also be used in education, national defense, business, and recreation. With very little modification the game would have the information and mechanics necessary to educate about P-LEO satellite constellations, or to model the attack and defense of P-LEO constellations between two entities such as nation states. With slightly more gamification the game could become a recreational game for satellite enthusiasts as well. The author of this thesis has co-authored several papers [7, 13] and contributed to a report [6] on the framework and the demonstrative game Sat-Tycoon.

## 1.7 Outline

Chapter 2 covers related work and details how each work compares to the framework. Chapter 3 introduces the more technical details of the client/server architecture, setting the stage for chapters 4 and 5 to lay out the provided human and AI clients built in JavaScript/React and Python respectively. Chapter 6 covers what makes the framework a good tool for researchers. Chapter 7 wraps everything up and discussions limitations of the framework and potential future work.

## Chapter 2

### Related Work

This framework is not the first Multi-Agent RL (MARL) framework to be developed; however, the way in which it operates and handles MARL may be quite novel. Other frameworks focus on being able to universally handle as many types of MARL games as possible, while SatTycoon’s framework is purpose-built to handle real-time asynchronous games. Another major difference is that this framework is designed to support playing with human players. Many if not all frameworks opt to use some kind of proxy or surrogate in place of a real human player for efficiency.

#### 2.1 OpenSpiel

OpenSpiel [9] provides not only a framework for testing different algorithms but also for writing different games to test algorithms in. The games are written in C++ and wrapped in Python, and the algorithms are in both Python and C++. Games in OpenSpiel are represented as mostly Extensive-Form Games (EFG), but may also occasionally be represented using Markov Decision Processes (MDP). EFGs are explained in more detail in Subsection 6.1.2, and Markov Decision Processes are explained in Subsection 6.1.1.

#### 2.2 PettingZoo

PettingZoo [18] is a library of environments inspired by OpenAI’s gym that was created with the goal of having a single standardized way to handle MARL. To this end, the PettingZoo developers wanted the PettingZoo API to be as familiar to gym’s API as possible. The use

of PettingZoo was considered for Sat-Tycoon; however, it was determined that for purposes of reaching a wider audience, Sat-Tycoon would use OpenAI’s gym instead. In addition, Sat-Tycoon is not a traditional MARL game, being real-time asynchronous in nature. This means that it is not ideal for a synchronous environment like PettingZoo would provide. Given that neither PettingZoo nor OpenAI have solutions to the real time asynchronous challenge, it would need to be addressed regardless of which API was chosen. In PettingZoo, the agents step sequentially using the agent environment cycle games model.

### 2.2.1 Agent Environment Cycle

The agent environment cycle (AEC) is a sequential model in which agents take turns seeing their observations, taking actions, and receiving rewards. In AEC models, the environment acts as an agent itself, its actions representing the updating of the environment in response to agent actions. This sequential model results in a turn-based game, which is directly in conflict with the real-time asynchronous nature of Sat-Tycoon.

### 2.3 Dopamine

Dopamine [4] is a tensorflow<sup>1</sup> based framework focused on being simple and compact, so that anyone can pick it up and use it regardless of experience level with RL. It initially only focuses on the Arcade Learning Environment [1], but the Dopamine team has plans to expand it later. While there is no clear statement on MARL support, with its focus on simplicity, compactness, and reliability, it is not likely to support MARL without modification. Sat-Tycoon boasts a simple API, but it is almost definitely not a simple, compact framework like Dopamine.

<sup>1</sup><https://www.tensorflow.org/>

Table 2.1: Table comparing different RL & MARL frameworks

Environment	MARL	Async	Real-time
Sat-Tycoon	Yes	Yes	Yes
PettingZoo	Yes	No	No
OpenSpiel	Yes	No	No
Dopamine	No	No	No

## 2.4 Related Aerospace Gyms

Similar aerospace reinforcement learning environments to Sat-Tycoon have been made, but they do not have the same breadth and complexity as Sat-Tycoon. Being single player, single agent while also having a smaller scale, more detail-oriented focus.

### 2.4.1 Deep Replacement: Reinforcement learning based constellation management and autonomous replacement

The Deep Replacement [8] environment focuses on monitoring the health of a constellation and optimally replacing damaged satellites. This is more focused on the intricate details of Satellite management, unlike Sat-Tycoon, which puts the focus on the construction and management of an entire constellation including not only the satellites, but also the ground stations and company finances.

The actions for the Deep Replacement environment are discretized into a set of finite discrete actions. They are “No Action”, “Build Piece Parts”, “Build Components”, “Build Subsystems”, “Build S/C”, and “Launch S/C”. These are more focused on the micromanagement of actual satellites than the actions available in Sat-Tycoon’s much broader focus on the construction and management of an entire satellite constellation.

### 2.4.2 StarCraft II

Following the development of the StarCraft II Learning Environment (SC2LE) [19], StarCraft II has become a popular environment for training RL agents. StarCraft II is provided to agents as an environment via its programmatic API, and environments such as PySC2 are built on top of this API.

#### PySC2

PySC2<sup>2</sup> is DeepMind’s<sup>3</sup> open source Python environment for RL agents provided as part of SC2LE. It provides action and observation spaces, as well as a handful of example agents

---

<sup>2</sup><https://www.deepmind.com/open-source/pysc2>

<sup>3</sup><https://www.deepmind.com/>



including a random agent. Included in the environment are mini games, challenges, and visualization tools to assist in understanding the agent’s capabilities in the environment. The provided mini-games and challenges allow developers to train agents on smaller individual objectives in the game instead of an entire match of StarCraft II. These can be objectives such as building a unit, traversing the game map, or collecting a resource. Such mini-games and challenges would be a great addition to the provided Sat-Tycoon game’s environment and are discussed further in Section 7.2.

## 2.5 Multi-Agent Reinforcement Learning

Multi-Agent RL is a subset of RL in which multiple RL agents learn together. This is usually done with multiple agents in a single gym, but in the case of Sat-Tycoon, which is a real-time asynchronous game, it must be done using multiple gyms connected to a single game.

## 2.6 Multi-Agent Reinforcement Learning Gyms

There are several MARL gyms already, and many of them share similarities with Sat-Arena. However, Sat-Arena meets some very specific needs not found in these other gyms. Namely, very few of them support human interaction and none of them support real time asynchronous play.

### 2.6.1 TanksWorld

TanksWorld [15] is a multi-agent environment in which two teams of  $N$  tanks compete in a battle. Unlike Sat-Tycoon, in TanksWorld states are represented using 128x128 4-channel images. The four channels are position and orientation of allies, position and orientation of visible threats, position of neutral tanks, and position of obstacles. Agents in TanksWorld have three different continuous actions at their disposal. They can adjust their velocity, turn their tank, or shoot their tank’s gun. Tanks in the TanksWorld environments may communicate with teammates, and the environment includes the communication range between teammates. Teammates can then inform each other of which opponents are in their field of view. This creates a cooperative dynamic between agents, encouraging teammates to manage their distance from

each other to effectively communicate vital information. The TanksWorld environment models uncertainty with known-unknowns and unknown-unknowns in mind. To accomplish this, TanksWorld provides several modifiable parameters to change things like the range a tank can observe around itself, the range at which a tank can communicate with teammates, and even the quality of communication which can occur. Sat-Tycoon is a partial information game with uncertainty built in at a fundamental level. Agents are completely isolated into their own gyms, and are unable to perceive what moves their opponents may be ready to make. This can result in situations where an ideal orbit shell location is taken, or an opponent drastically lowers their internet price. Sat-Tycoon also allows agents to communicate, albeit in a much more advanced form. Agents in Sat-Tycoon can communicate via text messages, and to any other players or agents in their lobby. Unlike TanksWorld, agent communication in Sat-Tycoon is not restricted to teammates. Sat-Tycoon also allows for human players to join and play in games directly, where TanksWorld only allows for human surrogate policies using behavior cloning [10]. Human surrogate policies are generally used because it is not feasible for a human player to sit down and play the number of games it would take for an agent to be trained, and even if they could it would take an unacceptable amount of time. However, there is no reason that Sat-Tycoon would not be able to also support human-surrogate policies as well, allowing for swift agent-only games.

### 2.6.2 Battlesnake Challenge

Battlesnake Challenge [5] is an AI gym for the game Battlesnake, which is a multiplayer extension of the classic game Snake. In Battlesnake, multiple snakes compete against each other for food while trying to avoid being eliminated. Snakes are eliminated either by not collecting enough food and starving to death, or by being eaten by a larger snake. Battlesnake Challenge is an agent-agnostic, heuristics-agnostic RL framework that allows developers to design their own RL agents and algorithms then showcase them in an online competitive environment. The Battlesnake framework offers both an offline environment for agent training and an online environment for agent competition. Sat-Tycoon does not have different modes, but the framework can be used both online and offline. Sat-Tycoon can support high speed local offline training by

utilizing its conveniently separated game logic with an offline gym environment. Battlesnake Challenge utilizes “Human In The Loop” (HIL) methods, allowing the use of human knowledge to assist in training an agent. This knowledge is provided either through real-time observation and feedback, or handcrafted rules to guide and alter agent behavior. Like human surrogate policies, this is another alternative to real-time human players playing alongside agents. Sat-Tycoon does not currently support this, but it could be added in future work.

## 2.7 Competitive Multi-Agent Environments

Table 2.2 shows a comparison of key characteristics of different competitive multi-agent environments with the Sat-Tycoon environment. They are compared based on player count, whether or not they are competitive or cooperative environments, whether or not they feature online networking, whether they have asynchronous turns, and whether or not they are real-time environments.

Of the reviewed environments, only Sat-Tycoon, NeuralMMO, and AIRPG implement networking code. Sat-Tycoon and AIRPG support network play, while NeuralMMO appears to only support a visualizer over the network. Sat-Tycoon and NeuralMMO both utilize WebSockets, allowing for the use of web browsers. Since AIRPG utilizes regular TCP sockets, it does not natively support modern web browsers. However, it appears to provide a Python GUI client for the game. Battlesnake Challenge uses offline training, but then provides an online environment for the trained agents to compete in.

Of the reviewed environments, NeuralMMO has the highest max player count with millions of agents playing at the same time. AIRPG did performance testing with 8 agents, but does not mention how many players their environment can handle at once. Battlesnake Challenge tested their environment with 6 agents but also does not make mention of a maximum player count.

### 2.7.1 NeuralMMO

NeuralMMO [17] is a large scale PettingZoo compliant MARL environment, inspired by the MMORPG genre of video games. The framework supports up to 1024 concurrent agents on

maps up to 1024x1024 in size, making it much larger in scale than the Sat-Tycoon game is intended to be. NeuralMMO is not just large however, it also features ample complexity for agents. The game mechanics offer complexity in the form of resources that deplete at each timestep, combat between agents, skills and abilities to level up, scripted non-player characters to interact with and fight, and equipment to wear. The environment also offers reward complexity through in-game achievements such as exploring the entire map or defeating X number of players. The provided gym environment for Sat-Tycoon features a rather simple reward function, but the game offers ample opportunity to implement a complex achievement system, and such achievements could easily be implemented and tracked on the server.

### 2.7.2 Online RPG For Reinforcement Learning

AIRPG [12] is another environment inspired by MMORPG games. In the same vein as NeuralMMO, agents in AIRPG have finite resources on a vast shared map. Like Sat-Tycoon, AIRPG uses OpenAI Gym clients for RL. AIRPG also appears to use a similar server authoritative architecture to Sat-Tycoon where the server handles data management and game logic for the clients. It also uses concurrency to handle the different loops required on the server, one for game logic, one for network communication, and one for enemy AI. Their client is based on OpenCV<sup>4</sup>, and acts as a GUI to manage rewards. The developers of AIRPG claim the proposed environment is compatible with OpenAI Gym, but it is unclear if it is using OpenAI gym, or how much the client actually does with regard to the gym. Their environment may be running locally on the server, and the client may simply be a visualizer. The AIRPG paper mentions

---

<sup>4</sup><https://opencv.org/>

Table 2.2: Table comparing different competitive multi-agent environments.

Environment	Player Count	Cooperative	Online	Async	Real-time
Sat-Tycoon	4	No	Yes	Yes	Yes
NeuralMMO	Millions	Yes	No	No	No
AIRPG	8	Yes	Yes	No	No
Battlesnake Challenge	6	No	No	No	No
TanksWorld	10	Yes	No	No	No

using threading for concurrent programming, but it is unclear how they do so without the interference of the Global Interpreter Lock. Data between their threads is also being shared on a queue, which could just be a coincidence, but is a popular way of sharing data between multiple processes in Python so it is possible that they meant to say “process” instead of “thread”.

## Chapter 3

### Client/Server Architecture and Model/View Separation

#### 3.1 Clients

Sat-Tycoon has a client agnostic architecture that allows any client using the WebSocket<sup>1</sup> protocol to connect to the server and play. For human players who do not wish to develop a client, a graphical client has been developed and provided using JavaScript and React<sup>2</sup>. For Reinforcement Learning developers, a non-graphical OpenAI Gym client has been developed in Python that utilizes the Sat-Tycoon API. Sat-Tycoon also allows any client utilizing WebSockets to connect to and play the game using the provided custom API. Sat-Tycoon was developed this way to facilitate its use as a research environment, allowing for the development and use of many different AI agents.

#### 3.2 Server

Sat-Tycoon is hosted on an authoritative Python WebSocket server. It handles all data management and game logic for players. The server does not distinguish whether a player is a human or an AI agent, and is client agnostic.

##### 3.2.1 Lobbies

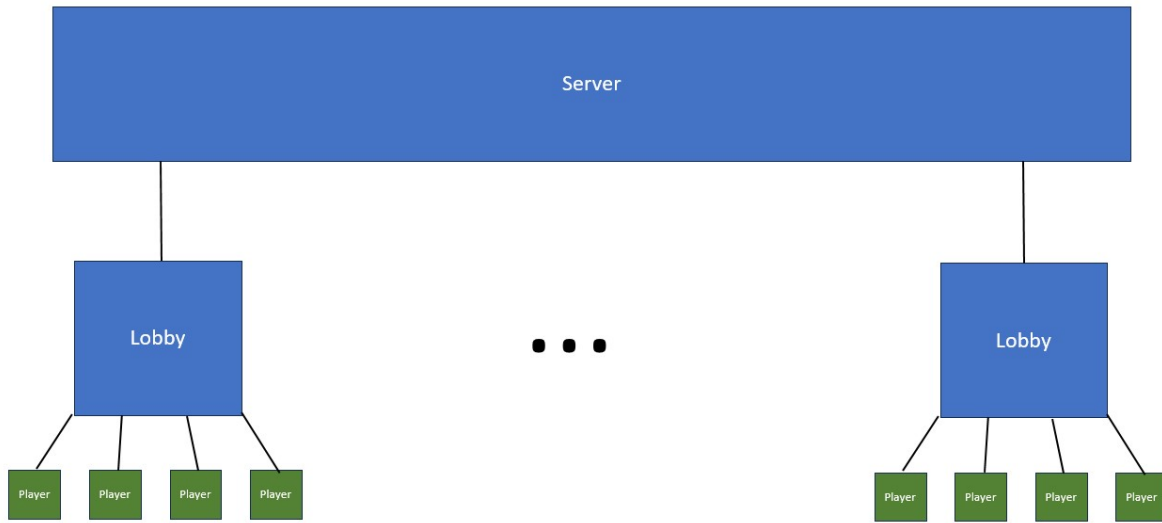
The Sat-Tycoon server organizes games into lobbies which manage the user objects associated with clients connected to the server. This structure can be seen in Figure 3.1. Upon connecting

---

<sup>1</sup>[https://developer.mozilla.org/en-US/docs/Web/API/WebSockets\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API)

<sup>2</sup><https://react.dev/>

Figure 3.1: The server organizes players into lobbies.



to the server for the first time, a client must register a name and the intended lobby it would like to create or join. Lobbies are a class object written in Python on the server that facilitate the ability to play multiple games of Sat-Tycoon at the same time. Lobby objects are stored in a global dictionary on the server, so that it can access any lobby at any time. Due to being stored in a dictionary using their name as the key, lobby names must be unique. Each lobby contains a Game class object that handles running that lobby's game loop. This means multiple games can be running at the same time. A lobby has functions to add a player, remove a player, broadcast a message to all player clients in the lobby, start a game, and toggle whether or not a game is being played. The current implementation is rather basic and could easily be improved by providing players with a list of existing lobbies they can join, as opposed to simply asking for a name string. Lobbies could also be upgraded to allow private lobbies, where a player can provide a password string to limit who can join.

### 3.2.2 Users

When a client connects to the server, a User object will either be created for it or assigned to it depending on if a "User" object with the requested player name already exists on the

server and is not in use. This User dictionary object is stored and used by the server in a Python dictionary with other User objects. A User consists of the client's chosen name, the client's unique WebSocket connection, an online status, and the name of the lobby the client has joined. This user object is separate from the player object stored on the game state, which is identified using the chosen name and lobby values stored on the user object. This separation of players and users allows for future development of consistent user accounts, and the ability to join and leave different lobbies.

### 3.2.3 Multiprocessing

There are two concurrent loops running on the server during a game: the networking loop and the game loop. To implement these two loops, multi-processing was used. The game loop and the networking loop are separated into two different processes and communicate using a queue. The networking process uses an asynchronous loop to handle many different clients at once. The alternative to multi-processing is multi-threading; however, due to Python's threading implementation this may not be the ideal choice when it comes to concurrent programming. This is because Python's memory management is not thread safe, so it has to use the Global Interpreter Lock<sup>3</sup>, or GIL for short. This single-lock implementation increases Python's single-threaded performance, but may introduce locking inefficiencies in situations like multi-threading. This is because the GIL can effectively prevent threads from executing simultaneously and can cause Python threads to execute in serial, not in parallel.

This means it is usually safer to use multiprocessing for concurrency in Python; however, this comes with its own challenge. Processes in Python cannot share data between each other directly. This is solved by utilizing a "Manager" from Python's multiprocessing library to create shared variables to share data between processes. The game loop updates a dictionary called the game state with any information clients may need, and this dictionary is shared via a multiprocessing manager. When a client requests game data, the server process checks this shared dictionary for the data it needs.

---

<sup>3</sup><https://realpython.com/python-gil/>



### 3.2.4 Game Logic

Sat-Tycoon's game logic is separated from the networking logic in order to facilitate the ability for the game to be run as quickly and efficiently as possible locally. The Game class contains the game logic, the game clock and several functions to facilitate calculating and updating the global population, player customers, and player revenues. These functions are called in the game loop at the appropriate time intervals using the game clock. A local client could opt to use the entire game loop and adhere to its clock or just use the game logic and simulate the passage of time in the game itself. A major advantage to this design is the ability for future researchers to develop their own game logic and API calls to play a different game entirely.

#### Game Clock

The game clock runs on a simple loop timer counting the days and months of the year. This loop is timed using the monotonic clock instead of wall time, so that it is not affected by potential variations. This ensures that the loop runs for a stable duration. To simplify dates and calculations months have been normalized to be 30 days long. Each iteration of the loop currently spans two days, however this number is easily adjustable allowing the pace of the game to be sped up or slowed down as necessary. The date is returned in a UTC timestamp using milliseconds so that the date is flexible and robust, allowing calculations to be done with it if necessary while also allowing it to be converted into a string for readability. On the provided React client, this UTC timestamp is interpreted as a string for human players using the JavaScript Date object however something like an AI agent may not need to convert it from milliseconds at all.

#### Game State

The game class maintains a Python dictionary called the game state that contains all of the data needed to see a snapshot of the game at any time. Updates happen to the game state at several different points in the game. Most notably when the date is updated each month, when a player makes an action, and when customer, population, and revenue data is updated each

month. The game state is important in facilitating an OpenAI gym in providing data to be used as the observation state for the gym environment. It also packages convenient snapshots of the game for logging purposes.

### 3.2.5 Ending the Game

The game ends after a predetermined number of years, currently set to ten years. When the game ends, the game loop will update a Boolean game state variable called “ended” that signifies the game is over, before terminating the game loop. The next time a client requests the date, they will be informed that the game has ended. After receiving notification that the game has ended, the React client will redirect the player to the scoreboard and display score data for each of the players. Sat-Arena, if it still has timesteps remaining will call the reset function. This will reset the gym’s observation space, disconnect and reconnect to the server, and start a new game.

## 3.3 Multiprocessing

The server uses multiprocessing in a few different ways, the most significant of which is to facilitate the use of WebSockets to communicate with clients asynchronously. In order for the WebSocket server to handle multiple different clients at once, it uses an asynchronous event loop with the library `asyncio`<sup>4</sup>. The main event loop is called the server-loop, and this is where “tasks” are “scheduled”.

## 3.4 Server Authoritative Architecture

The framework implements a server authoritative architecture. As an authoritative server, it manages and maintains the game state for the clients. Clients cannot modify the game state, but instead are required to make a request to the server to interact with it. In this way, clients act more like a video game controller between the player and the game.

---

<sup>4</sup><https://docs.python.org/3/library/asyncio.html>

All game logic is performed server side, and all game clients pull from the same central data set hosted on the server. The clients then maintain a local game state on their end using the data received from the server. This can be seen in Figure 3.2a. This helps to prevent clock and data synchronization problems between clients during gameplay. This also reduces cheating as a malicious agent will need to access the server directly in order to alter the game state.

The server receives requests to act upon the game state from clients as “actions.” These actions are formatted as JSON objects containing a plain English “action” string and a data payload. This exchange can be seen in Figure 3.2a. JSON was chosen because in order to be sent over sockets, and therefore over WebSockets, data must be serializable. In this JSON object the action string informs the server about what the client would like to do, such as build a ground station. The data payload contains parameters for that action, such as the latitude and longitude at which to build the ground station. To help create a more secure, robust environment the server maintains a set of acceptable actions. This allows the server to ignore malicious or incorrect requests from clients.

Server responses are formatted identically to actions, utilizing the same JSON structure. Upon receiving an action from a client the server will check against the predetermined set of acceptable actions to find which action it has received. If the action is not valid, it will send the client an error message. If the action is valid, however, then the server will validate the payload data before responding. Action validation involves making sure that the payload’s parameter values are legal within the bounds of the game as well as making sure that the action includes the correct parameters. For example, if a client has no money then they cannot purchase a ground station, and a client also cannot purchase a non-integer or negative number of ground stations. Invalid parameters such as these will also result in the client receiving an appropriate error message. The flow of an action through the server can be seen in Figure 3.2c. The flow of a response being processed by a gym client can be seen in Figure 3.2b.

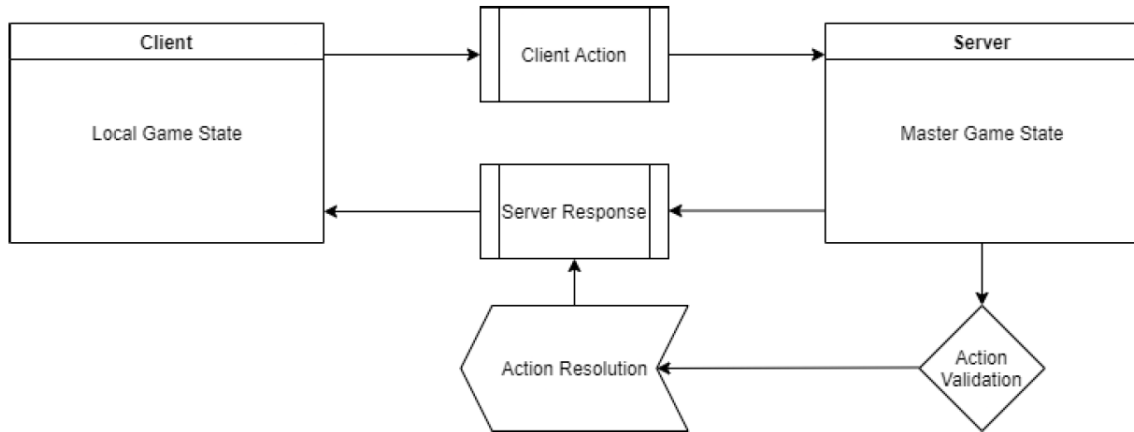
The server handles actions on a first come, first serve basis. This means if an agent sends an invalid action, the server sends a response informing them and immediately starts processing the next action in the queue. The easiest way to think about this is that the agent essentially loses their spot in line and has to hop back in at the back of the line. This maintains the

real-time nature of the provided Sat-Tycoon game environment. If an opponent isn't faster, or doesn't have an action waiting, the agent may still have their action act first if they put it back in fast enough. This can be easily changed by simply changing how the server handles actions, such as giving it a player order so that it checks to make sure the next action processed is the player whose turn it is in the provided order. It is important to note this would make the game turn-based instead of real-time.

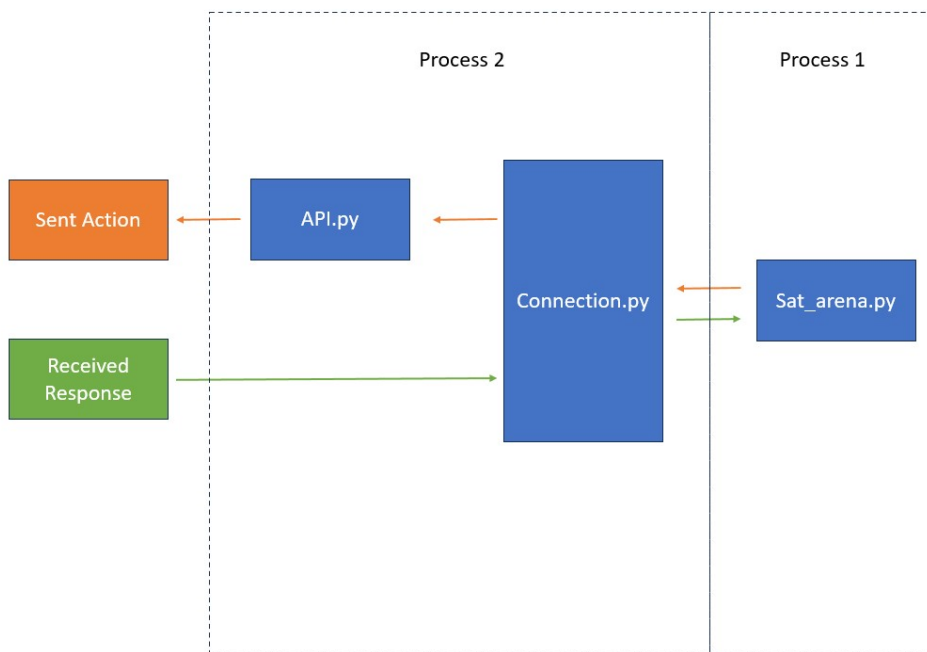
Upon performing an invalid action and receiving an error response in return, the game state will remain unchanged by that invalid action, as the action is not processed. However, opponent actions may continue to occur. The client will continue to receive these environmental updates based on opponent actions as well as normal scheduled environmental updates such as population changes. From the perspective of the client, their game state continues to change along with the game as if they had not made an action. It is then up to the player to handle how to respond to this situation. A human player will likely just correct the action and try the same action again, but an AI agent may move on and try something completely different.

Figure 3.2: The exchange of data between the server and a gym client, and the internal data flow for each.

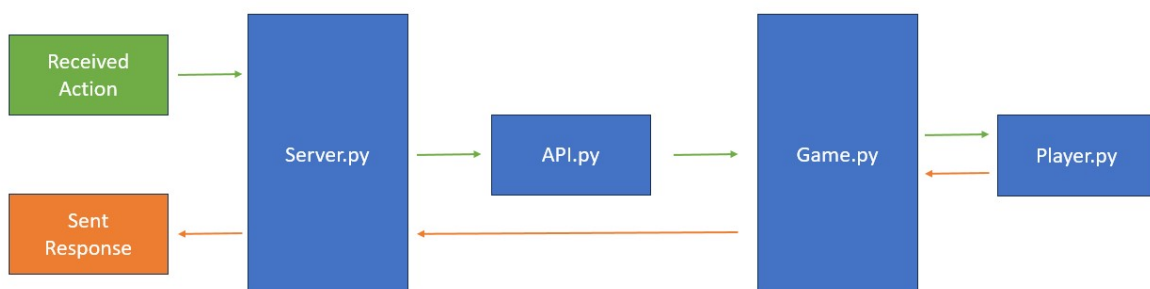
(a) Example of client/server communications, with action validation.



(b) The flow of an action through the Sat-Arena gym client and the reception of a response.



(c) The flow of an action through the server and the return of a response.



## Chapter 4

### The React Client for Human Players

The provided human player client for Sat-Tycoon was developed using the React library for JavaScript. React is a graphical user interface (GUI) framework primarily used to develop web applications. This gives a couple distinct advantages over using a more traditional game engine like Pygame<sup>1</sup> while also meeting Sat-Tycoon's specific needs better than a web-based game engine like Phaser<sup>2</sup>. This is because Sat-Tycoon does not require many features offered by a game engine such as physics, animations, and controller interfaces. Sat-Tycoon is an economic game, and therefore very data heavy. It benefits most from robust user interfaces such as those made easier to create by React. Using React also makes it easy for the client be hosted online, allowing players to connect to and play Sat-Tycoon from anywhere using a web browser.

There are several competitors in the web app development space, but the two most popular options are React and Angular. React is a JavaScript library developed by Facebook, and Angular<sup>3</sup> is a framework developed by Google. Both are built for the purpose of developing robust, flexible web interfaces. React and Angular are quite similar, with the major difference being that React is a library and could technically even be used in an Angular app. However, a key advantage of React is its use of the Document Object Model, or DOM for short. The DOM is essentially a tree of the different documents that make up a single page application. React uses what is called a Virtual DOM, where-as Angular uses the real DOM. Using the virtual DOM, React does not need to update the entire page when updating data. The virtual DOM acts as a kind of blueprint, where it can figure out what needs updated and how before

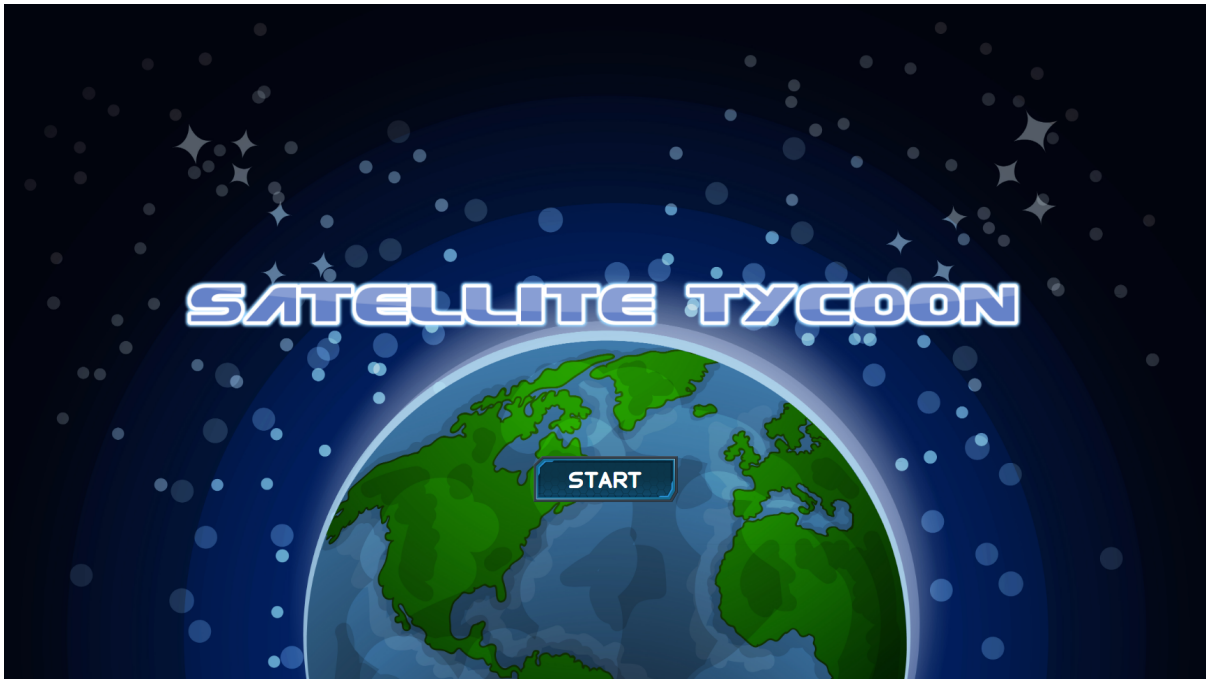
---

<sup>1</sup><https://www.pygame.org/>

<sup>2</sup><https://phaser.io/>

<sup>3</sup><https://angular.io/>

Figure 4.1: The initial title screen for the React player client.



making any changes to the real DOM. This gives React meaningful performance and stability advantages, as updating fewer moving parts results in faster speeds and fewer points of failure.

## 4.1 Player Login

Upon initially connecting to the server, the client displays a simple login screen requesting players to provide the name they wish to use to represent themselves in the game, and the name of the lobby they would like to join. Two text input boxes are provided to enter this data. This screen is followed by a lobby screen that displays the players who have joined the game. From the lobby screen, players can click the ready button to signal that they are ready to play the game. When all players have readied, the game will begin.

### 4.1.1 Zero Turn

Players begin in an optional “zero turn” shown in Figure 4.4 where they can set up an initial constellation, and are given a set amount of funds. The zero turn is designed to act as a kind of tutorial, or soft entry into the game mechanics of Sat-Tycoon. As such, it offers three different experiences to players upon entering: Beginner, Experienced, and Professional. Players that

Figure 4.2: The login page allows players to choose a name and lobby.

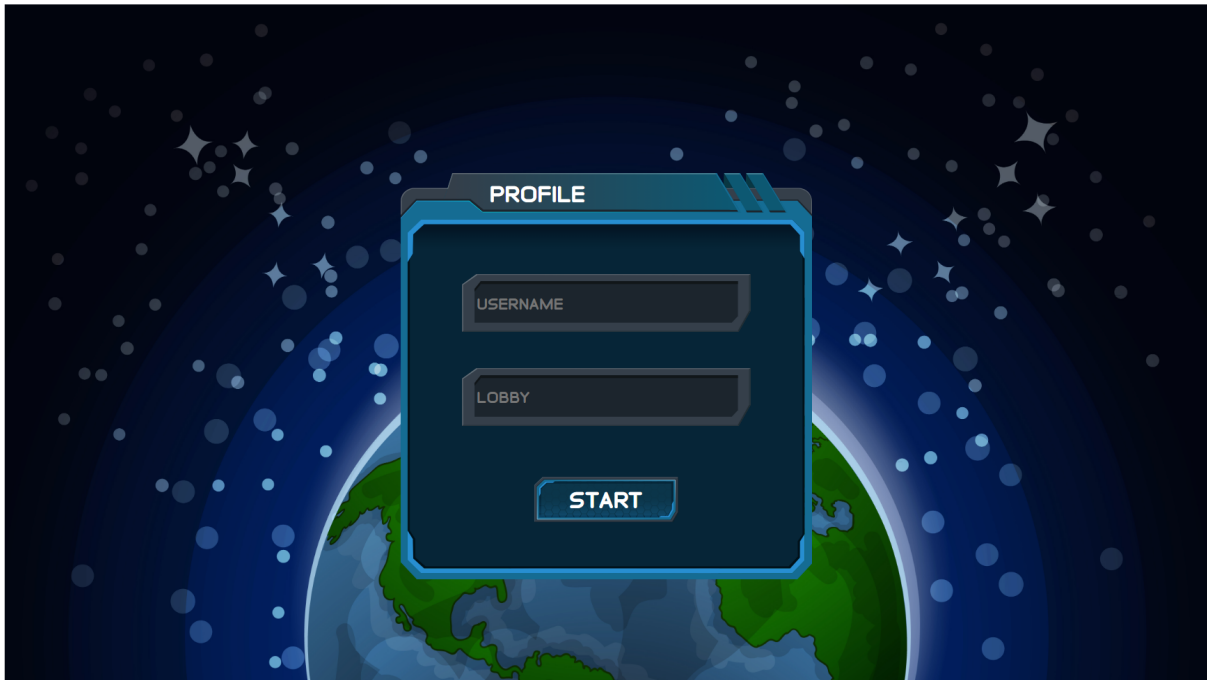


Figure 4.3: The lobby page shows which players have joined the game, and lets a player signal they are ready to play.

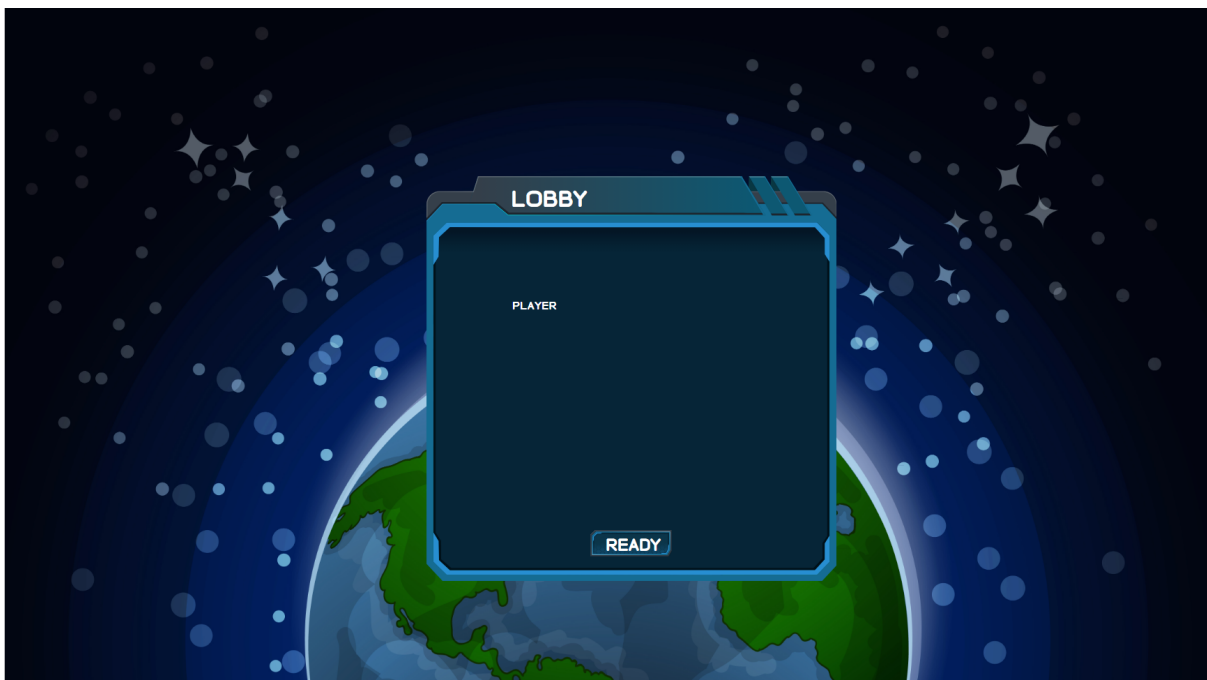




Figure 4.4: The zero turn provides a tutorial experience for players who may be new to the game.



select the beginner option will be presented with brief explanations of each game mechanic, and how to interact with the game. Players who select experienced will just be given the ability to make these initial game decisions, and those that choose professional skip the zero turn entirely.

## 4.2 Components

React structures an app into multiple purpose-built “components”. Components are reusable pieces of code that are organized in a parent-child object oriented structure. There are two types of components in React, “class” components and “function” components. Sat-Tycoon uses class components.

### 4.2.1 Footer

The footer in Figure 4.5 is a persistent UI feature that can be seen on any page of the game. It serves to always provide players with a clear picture of how many customers they have, how much funding they have left, which technology they have selected, and what their name is.

Figure 4.5: The footer provides a clear picture of a player’s resources during gameplay.



Using the technology buttons on the footer, players can change which technology they have selected from any page.

#### 4.2.2 Multi-Panel

Instead of making players navigate back and forth between different pages for the many different elements of Sat-Tycoon, the user interface was designed with a “multipanel” as seen in Figure 4.6, that features tabs to switch between several different panels. Each panel represents a different element of Sat-Tycoon, and as a player switches between them the main gameplay area changes as well. For example, the Earth panel will change the main game screen to display a map of the Earth, while the Space panel displays the SpaceDeck.

#### 4.2.3 Earth Panel

The Earth panel in Figure 4.7 is the initial panel displayed when the game starts. It contains game elements related to the Earth, namely ground stations and customer internet prices. There are text input boxes for manual entry of latitude and longitude, but given that not all researchers or players may have an aerospace background, a list of predetermined locations is also provided.

#### 4.2.4 Space Panel

The Space panel in Figure 4.8 handles the purchase of Satellites and displays an animated map of the coverage a satellite will provide after being launched into a certain orbit plane. It is complimented on the main screen by the SpaceDeck, which allows players to select orbit planes for either launching satellites to or viewing their coverage.

Figure 4.6: The multipanel serves to help players access multiple areas of the game from one location.



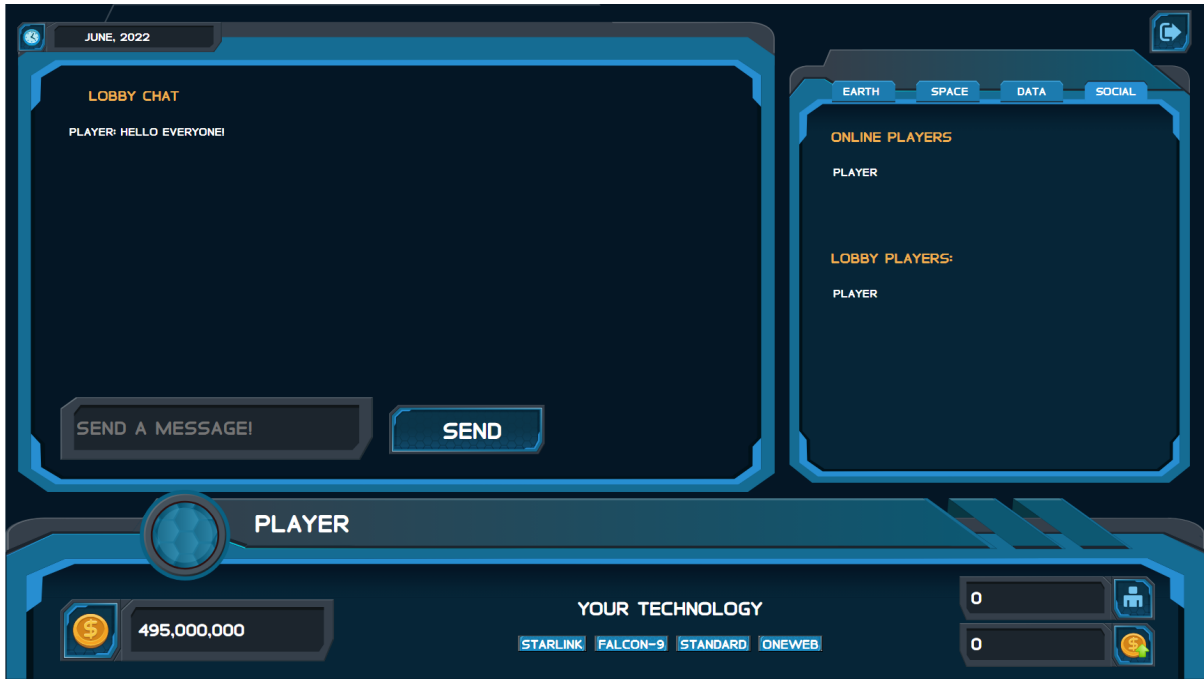
Figure 4.7: The Earth panel provides access to features such as the world map and groundstation placement.



Figure 4.8: The Space panel provides access to features such as the Space Deck and launching satellites.



Figure 4.9: The Social panel provides a place for players to interact with each other directly.



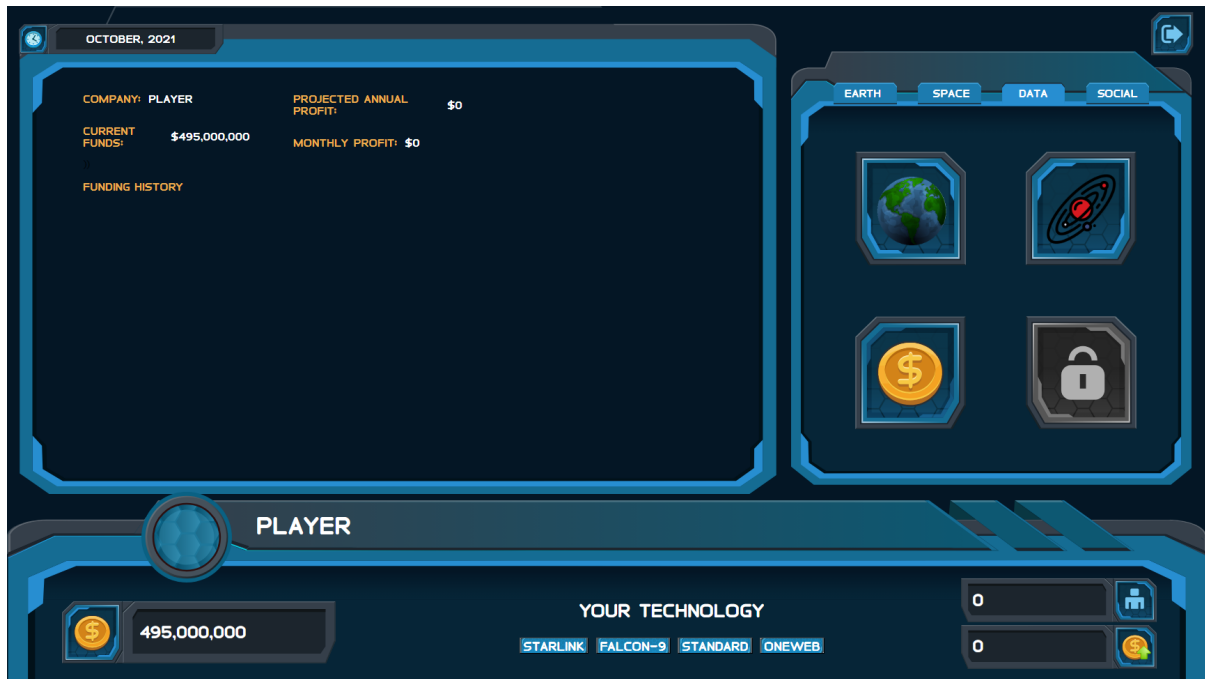
#### 4.2.5 Social Panel

The social panel in Figure 4.9 is an early implementation of a more interactive player experience in Sat-Tycoon. It has a list of all players online in all lobbies, as well as a list of players online specifically in the current player's lobby. It also has a chatbox where a player may chat with the players in their lobby. It would be easy in the future to add more features to the chat box, such as the ability to create and join chatrooms, as well as the ability to privately message a single user. The chat box can also function as a place for system broadcasts from the server to be displayed, although in such a case a visual cue for players not currently looking at the chatbox may be necessary to inform them that a broadcast has arrived. Another option would be to put a smaller chatbox in the footer.

#### 4.2.6 Analytic Panel

The analytic panel in Figure 4.10 shows players useful information about the data visible to them in the game. It is currently a proof of concept that contains three sections: Earth, Science, and Finances. The Earth section shows data related to the Earth panel such as customer history, current customers, and internet price history. The Space section similarly shows data related to

Figure 4.10: The Analytic panel gives players important statistical data about the game.



the Space panel. The finance section is self explanatory, showing data about monthly revenue, total funds, and other economic information.

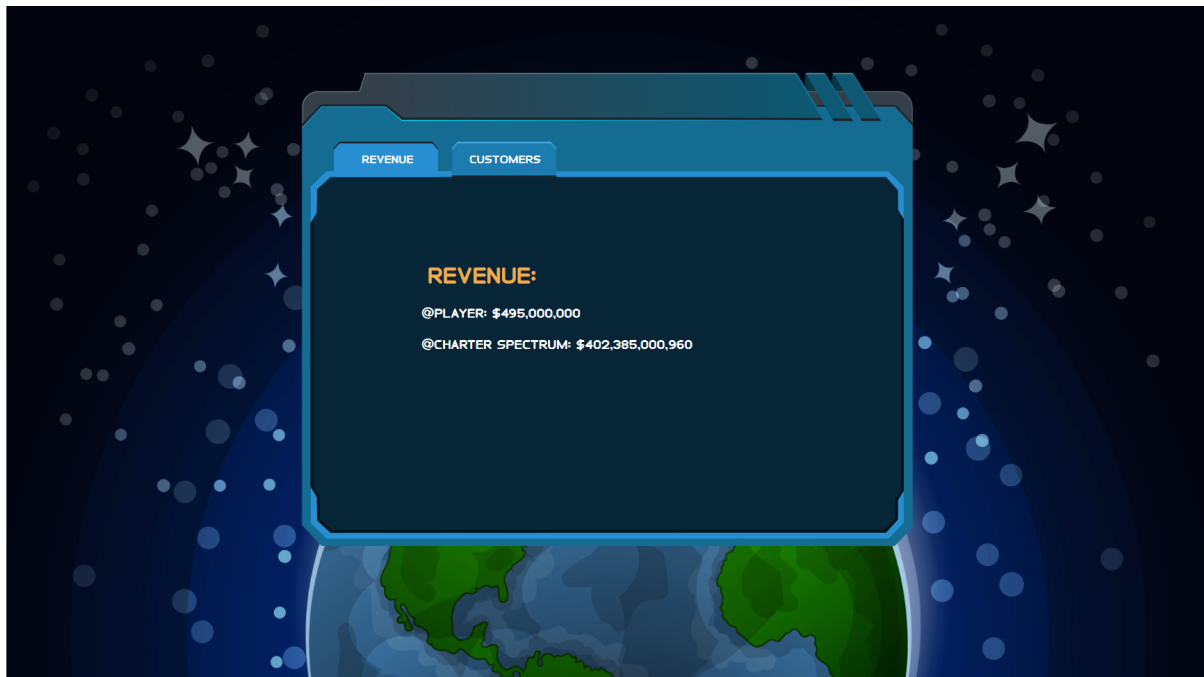
#### 4.2.7 Scoreboard

After a game has been completed, players are presented with a scoreboard page as seen in Figure 4.11 that showcases how they performed compared to competing players. It features tabs for customer count and revenue, but may easily be expanded with more values to be compared in the future.

#### 4.3 Redux State

The parent-child structure of React means that it can be difficult when you need to share data back up the hierarchy, from a child to a parent. It can also be difficult to share data from one child to another child whom may not be a direct sibling. Sat-Tycoon solves both of these challenges using Redux. Redux is a state manager for React, allowing the application to instead store data in a Redux state on a Redux “store.” Components are then connected to Redux so

Figure 4.11: The scoreboard displays how well players fared against their competitors after a game has ended.



that they may access the different Redux stores. This provides the client with a sort of global state, while also cleanly organizing all of Sat-Tycoon’s game state data in one place.

#### 4.4 Networking

Sat-Tycoon is built with a server authoritative network architecture. The game logic and the user interface are hosted separately, divided into any number of connected clients and a single server. Connecting these clients to the server is the WebSocket protocol. The WebSocket protocol was chosen for its ability to easily allow web browsers to connect to the server alongside more traditional clients. In this architecture, the server maintains a “master copy” of the game state that only it can modify. When clients want to make an action in the game, they send a request to the server requesting an action be made on their behalf and receive the resulting game state changes back. This interaction is illustrated in Figure 3.2a. This makes it challenging to cheat in Sat-Tycoon as a client would need to access the server and modify the master game state directly to do so. It also prevents data synchronization issues that may be seen in a more distributed architecture like peer-to-peer.

#### 4.4.1 API

The framework's API is designed using the JSON data structure. When developing the API, a standardized object was necessary to represent the actions and their related information being sent between the server and clients. In addition, any data structure used to transfer data over a socket or WebSocket connection needs to be serializable. For this purpose JSON was chosen. The JSON data structure is widely accepted with many programming languages supporting it. It is nearly identical to a Python dictionary in structure, and it is trivial for the server to convert between the two.

The JSON API objects are designed with two primary elements in mind: The action and the payload. The action is a simple string object describing what action the agent wants to perform, such as building a groundstation ("build\_groundstation") or launching satellites ("launch\_satellites"). The accompanying payload is a dictionary describing the necessary data to perform this action. So the action to build a groundstation would have a payload including the "latitude" and "longitude" elements in its payload.

Every JSON response object sent back by the server also includes supporting elements, that may be of use to clients but are not required to play the game. These are a timestamp and a player name. The timestamp is a millisecond representation of the number of seconds passed since the 1970 time epoch. This allows the timestamp to be recognizable and usable by any programming language a client may be written in, and is the most robust, flexible option. The player's name is used to attribute the response to the client who made corresponding action. In cases where there was no corresponding action, like a date request or population update, the player is "system."



## Chapter 5

### The Sat-Arena OpenAI Gym Client

A client to allow a reinforcement learning agent to play the game using a custom OpenAI gym environment was developed using Python. The gym client runs entirely from the command line, and does not present the user with a graphical user interface.

#### 5.1 Simplified Environment

A simplified Sat-Tycoon gym environment is provided which reduces the action and observation spaces to Box spaces, representing the action and observation spaces as matrices. This simple gym is important in allowing more standard policies to be utilized with Sat-Tycoon. The stable baselines library in particular is compatible with the simplified gym allowing quick access to RL training and testing in Sat-Tycoon. To develop a more standard gym environment, a custom policy may be necessary requiring more overhead from a developer before training may begin.

The action space consists of all actions from the standard environment but compacted into a single box space, This box space contains values for the parameters of every possible action, as well as a number denoting which action has been chosen. The environment will use this chosen action value to determine which parameters to use, for example when building a ground station the quantity and internet price parameters go unused.

The observation space simplifies the original drastically, transforming it into a single linear array of all observed values. The observation data is stored in its native format, and the 2D arrays are flattened into 1D arrays that are then concatenated into the observation space array.

This method of linearizing the observation space is a dated technique, but will allow researchers to run more basic algorithms and policies such as those found in the Stable Baselines library on Sat-Tycoon.

## 5.2 Step

The step function serves to calculate the next state and reward for the agent. In order to do this, the step function takes an action from the agent. This action is parsed and used to generate a valid action to be sent to the server using the API. The step function in Sat-Arena also serves to handle interfacing with the networking API. At the beginning of each step, it checks for a connection process and whether or not there is a verified connection to the server. If either of these things are not there, it calls a connection handler function to initiate a new connection to the server. The step function then handles creating actions to send to the server as well as interpreting responses received from the server, all with the purpose of calculating the next state and reward.

## 5.3 Reset

The reset function sets all of the environment variables in the observation space back to their initial values, preparing the environment for a new game of Sat-Tycoon to be started. As such, it is called at the end of a game. This function specifically resets the data structures containing the observation variables, and clears out the game state. Since Sat-Tycoon is an online multiplayer game, the reset function also calls a function responsible for handling the network connection. This function is the same one used at the beginning of the step function. It checks for a connection process, and if one exists terminates it before creating a new connection process and starting a new connection with the server.

## 5.4 Timesteps

When training an agent using Stable Baselines, a number of Timesteps for which to run the gym needs to be provided. Stable Baselines will continue training the agent until these timesteps are exhausted, calling the reset function as necessary when a game ends.

## 5.5 Networking

Being designed for use with an online framework, Sat-Arena was developed with network connectivity in mind. Because many researchers use wrappers and other add-ons in their training methods for their agents, they often need a gym to allow for method ambiguity. The best way to allow for this is to make sure that the gym operates as closely to a standard gym environment as possible. For this reason, Sat-Arena is designed so that it can be installed and run like any other OpenAI Gym environment, and the networking functionality is an unobstructive part of the regular run of the gym environment.

### 5.5.1 The Connection Class

The gym connects to the server via WebSockets using a connection class that is initialized during the agent's first step function. This connection class has a member function called `connectToServer` that creates an asyncio event loop to run another member function called `websocketHandler` on. The `websocketHandler` member function is the core of the networking logic for the client. This is where the client makes the connection to the server using the "async with connect" loop. It is important that the gym environment be started first and call this code from inside of its loop as opposed to starting the networking code first and then initializing and starting the gym environment. When the connection is made first, and then the gym environment is initialized and started inside of the connection loop, it is harder for researchers to use things like wrappers because the gym environment is obfuscated.

### 5.5.2 Connecting

At the beginning of the step function, the gym checks if it is connected or not. If it isn't, it will connect to the server. After the initial call to create the connection, all connection related activities are handled in a separate process parallel to the gym environment. The agent will not send a ready signal to the server to begin playing until the indicated number of players in the environment's shared connection state dictionary are connected. This allows time for other agents or even human players to join the lobby with the agent before starting the game. The environment's step function uses the connection object's action queue to communicate actions with the server, and has a listener member function used to check the connection object's response queue for receiving responses back from the server.

## 5.6 Multiprocessing

The gym client utilizes multiprocessing to run the gym and a WebSocket connection to the server in parallel. When the gym enters the first step of the game, it runs a function to check to see if there is an existing connection process. If there isn't, it creates one. If there is, it terminates it and creates a new one. This new process runs a function to create a new connection class object, and then make a connection to the server using a member function. These two processes communicate between each other using a Python multiprocessing Manager that handles shared variables for them. There are three shared variables: The action queue, the response queue, and the connection state dictionary.

### 5.6.1 Action Queue

The action queue is used to share actions the agent wishes to take with the connection process. The agent puts an action to send on the queue, which is then removed from the queue and sent in the connection process.

### 5.6.2 Response Queue

The response queue is used to share actions the client has received from the server with the agent. The connection process puts a response on the queue when it receives it, and the agent takes the received responses off the queue in the listener function at the end of each step.

### 5.6.3 Connection State Dictionary

The connection state dictionary is used to share any variables or data related to the network connection, between the agent's process and the connection process. This includes things like whether or not the connection is verified, what host and port to use, the agent's name, which lobby the agent wants to play in, and how many players are currently connected.

## Chapter 6

### Sat-Tycoon as a Research Framework

#### 6.1 Supported Games

Sat-Tycoon is a real-time asynchronous game, and as such can be a difficult game to model. Sat-Arena has its own solution to this, however, the framework may support other, more common types of games. With some slight modification it would be possible to support models such as Markov Decision Process (MDP) or Extensive-Form Games (EFG). A guide on changing the framework's environment to support environments/games other than Sat-Tycoon can be found in Appendix A.

##### 6.1.1 Markov Decision Process

The Markov Decision Process is used to mathematically model partially random games in which ML agents are faced with a cost/reward optimization problem. According to the Markov Property, only the current state matters when calculating the next state. This means that only the immediately previous state matters when calculating the current state. This property is challenging to maintain for a MARL environment, where every agent is making actions that affect the environment, and therefore the next state. While it is certainly possible to maintain the Markov property in traditional, sequential MARL, it is much harder if not impossible in a real-time asynchronous environment like Sat-Arena.

### 6.1.2 Extensive-Form Games

An extensive-form game (EFG), is a game which can be described in its entirety using a tree. The tree should represent every player, their potential actions, their visible information, and the reward for each possible action combination. While these games are typically sequential in nature, they can also represent synchronous games. However, it would be infeasible to represent a game like Sat-Tycoon using EFG due to the difficulty in representing every possible state of the game.

## 6.2 Independent Gym Environments in Multi-Agent Learning

The framework includes a prebuilt OpenAI gym environment designed so that every agent is run in its own independent environment. This means that agents are completely isolated from each other, and there is no chance for information to be accidentally shared between agents. This allows for a pure hidden information environment.

Another advantage of the framework's independent gyms is the ability to take advantage of distributed computing. MARL agents do not have to be sharing an environment, therefore they do not even have to be on the same system. Agents can have full access to system resources without sharing between other agents.

## 6.3 Client Agnostic Design

As for clients, the framework is designed with a client agnostic design in mind. The server does not know what kind of client it is connected to, or who is on the other end. This means that a researcher or player can develop their very own Sat-Tycoon client to meet their specific needs. The requirements for a Sat-Tycoon client are simple: the client must connect to the server using the WebSocket protocol, and communicate with the server using Sat-Tycoon's JSON API calls. In the case that a researcher has built their own game in the likeness of Sat-Tycoon using the framework, they would of course use their own API calls instead.

Designing the framework to be client-agnostic has drawbacks, however. When designing new features for the game or framework, care needs to be taken that one particular client design

isn't being favored. For example, one should not design an API call that sends data in a specific way that one particular client needs it, but should instead send the data in a raw format that clients can then manipulate to suit their particular needs. When developing clients for a research environment in which humans and AI will be playing together, this is especially important to insure that the AI has access to the same information that a human player does.

#### 6.4 Possibilities

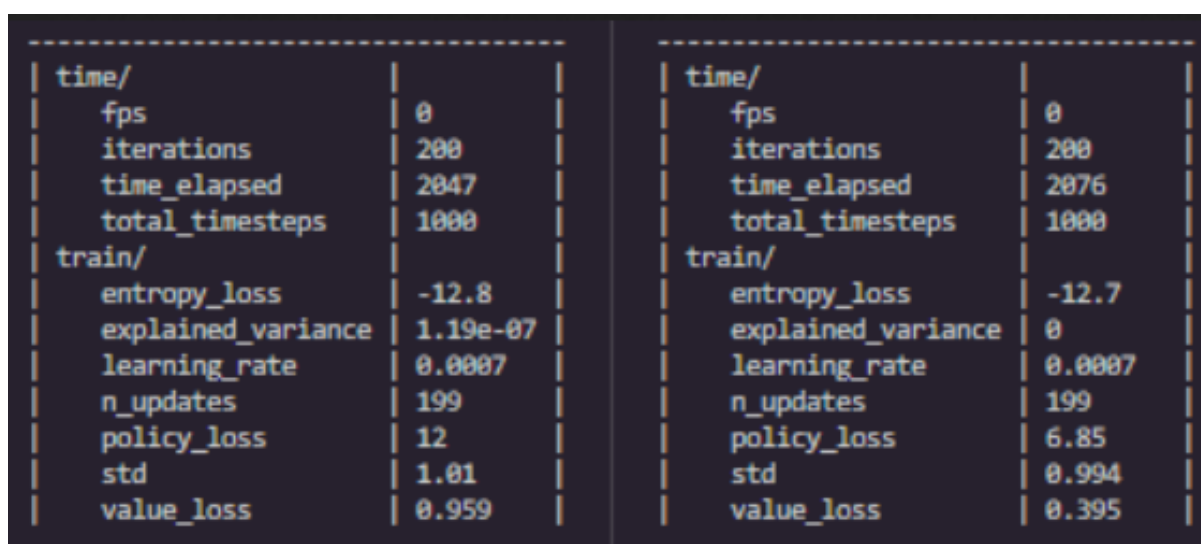
The major benefit to the framework being built this way is the near-infinite possibilities it presents to researchers looking to develop their own research tools or games. Any programming language that supports WebSockets can connect to and utilize the Sat-Tycoon server and framework; this makes the possibilities feel truly endless as there is even a WebSocket implementation for embedded systems in the C programming language. More information on developing a new game or environment for the framework can be found in AppendixA.

#### 6.5 Results

Figure 6.1 shows the output of a successful game of Sat-Tycoon between two agents using the Actor Critic algorithm A2C from the Stable Baselines 3 library to showcase the environment's compatibility with the library.



Figure 6.1: Output from a successful game between two agents using the A2C model in Stable Baselines 3.



Left Screenshot (200 iterations)		Right Screenshot (2076 iterations)	
time/		time/	
fps	0	fps	0
iterations	200	iterations	200
time_elapsed	2047	time_elapsed	2076
total_timesteps	1000	total_timesteps	1000
train/		train/	
entropy_loss	-12.8	entropy_loss	-12.7
explained_variance	1.19e-07	explained_variance	0
learning_rate	0.0007	learning_rate	0.0007
n_updates	199	n_updates	199
policy_loss	12	policy_loss	6.85
std	1.01	std	0.994
value_loss	0.959	value_loss	0.395

## Chapter 7

### Conclusion

This thesis presented a state-of-the-art research framework for asynchronous real-time multi-player games, both for human and AI players. This general purpose framework utilizes a robust, flexible JSON API over the WebSocket protocol to provide a client-agnostic environment ideal for research, business, education, or recreation. Utilizing a server authoritative architecture makes the framework robust and secure for AI to play without information leakage or cheating. The framework's simple API and separation of logic means it can be repurposed for any number of games, and with little modification supports many different kinds of games including MDP and EFG games. The provided Sat-Tycoon game provides an interesting playground for AI with rich, expansive action and observation spaces.

The following sections detail its known limitations and potential future lines of inquiry.

#### 7.1 Limitations

The current framework has several types of limitations related to reinforcement learning, design, hardware, and networking as detailed in the following subsections.

##### 7.1.1 Reinforcement Learning

While the framework supports reinforcement learning agents, it may not have all of the features that seasoned reinforcement learning experts may need. The provided RL gym does not maintain the Markov Property, for example. There may be changes that could be made to either the server, the game, or the gym to allow the framework to maintain the Markov Property for future researchers.

The framework also uses client-based gyms, which means there can be network latency involved when training or playing the game. A better idea may be to run the gyms on the server, and create clients that submit their agents remotely from the client's location to be run in a gym environment locally at the server's location.

### 7.1.2 Design

The framework was designed with a client-agnostic philosophy. While this offers many advantages, it also comes with drawbacks. This means that all design decisions, fixes, and feature additions have been made in a server-first fashion. This can make some things less straight forward, or off-load work onto the clients. For example, instead of sending geoJSON<sup>1</sup> data from the server to the React player client, the raw map data should ideally be sent instead and then converted on the client's end. This way, other clients are not forced to use the geoJSON format.

### 7.1.3 Hardware

The framework relies heavily on the hardware for scalability and performance. The better the server hardware, the more scalable and the more performant the game will be. This extends to clients, especially RL clients, which will greatly benefit from more computational power.

### 7.1.4 Networking

The game being networked inevitably means that there will be limitations involving network latency, packet loss, or even potential disconnections disrupting gameplay. This can be remedied by using localhost to play offline, or by developing an offline gym to utilize the game logic independently from the server.

## 7.2 Future Work

The following subsections will outline future work. This work includes improvements to the human player experience in the form of improvements to the React player client and a visualizer

---

<sup>1</sup><https://geojson.org/>

client, as well as persistent data storage on the server. It also covers improvements to the RL research experience which also benefits from the addition of a visualizer client and adding more complex gym environments.

### 7.2.1 Converting React Client to Functional Components

The current version of the Sat-Tycoon human player client in React uses React class components. While this works, and there is nothing inherently wrong with using this type of component it is outdated and the React community no longer uses them. It would be worthwhile to update the client to use functional components as recommended by the React team, which may even come with some improvements.

### 7.2.2 More Complex Gym Environments

The current provided gym environment is a simplified example designed to be compatible with the Stable Baselines 3 library of RL algorithms. It maybe worthwhile to develop a more complex environment to support more advanced RL algorithms and policies. Such an environment could use dict spaces for the action and observation spaces, utilizing a one-hot configuration for agents to decide which action to select.

### 7.2.3 Mini-games and Challenges for RL Agents

The provided Sat-Tycoon game is a very complex environment, and it may be fruitful to develop smaller challenges for AI to complete much like the ones that were developed for StarCraft II. Such challenges or mini-games could include things as simple as launching satellites, or more complex goals like attaining a certain amount of profit or subscribing customers in a certain region of the world for example.

### 7.2.4 Visualizer Client

A client designed specifically to visualize and present the game to spectators could serve many purposes. It can be useful for AI developers to be able to see exactly what their agent is doing in a game and observe behaviors that way. It can also be fun for human players to spectate

each other, or even to use such a visualizer client to display games on a stream or broadcast at an event for many people to see. This can make events where matches between different AI happen much more interesting for human attendees.

#### 7.2.5 Server-side Gym Environments

The development of a client capable of receiving submitted agents and sending them to the server to be run on a gym environment local to the server would greatly improve the efficiency of training. Network latency and response time would no longer be an issue and the client would simply act as a visualizer to display training data to the user. Using this client model, the agents could also maintain the ability to play online with humans if desired.

#### 7.2.6 Persistent Data

The current server implementation stores player data and game data in volatile memory. This means everything is stored in variables during the live run of the program, and as soon as the program ends or is restarted all of that data is lost. It would be rather straight-forward to implement a database system to store this data locally on the server allowing the framework to store and pull from persistent storage. This allows for great quality of life improvements to the framework, such as saving replays of games, maintaining persistent player accounts, maintaining player match histories, and even leader-board rankings for the best players or AI agents.

## References

- [1] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.
- [2] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.
- [3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.
- [4] Pablo Samuel Castro, Subhodeep Moitra, Carles Gelada, Saurabh Kumar, and Marc G Bellemare. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.
- [5] Jonathan Chung, Anna Luo, Xavier Raffin, and Scott Perry. Battlesnake challenge: A multi-agent reinforcement learning playground with human-in-the-loop. *arXiv preprint arXiv:2007.10504*, 2020.
- [6] Davide Guzzetti and Daniel Tauritz. Modeling Economic Competition in the Business of Mega-Constellations. Technical report, Auburn University, 2022.
- [7] Davide Guzzetti, Daniel R Tauritz, Rehman Qureshi, Cody Roberts, Manuel Indaco, Lucy Bone, and Emily Kimbrel. Satellite Tycoon: Modeling Economic Competition in the Business of P-LEO Constellations. *11th International Workshop on Satellite and Constellations Formation Flying*, 2022.

- [8] Joseph Kopacz, Jason Roney, and Roman Herschitz. Deep replacement: Reinforcement learning based constellation management and autonomous replacement. *Engineering Applications of Artificial Intelligence*, 104:104316, September 2021.
- [9] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. OpenSpiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.
- [10] Michael Laskey, Jonathan Lee, Roy Fox, Anca Dragan, and Ken Goldberg. Dart: Noise injection for robust imitation learning. In *Conference on robot learning*, pages 143–156. PMLR, 2017.
- [11] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.
- [12] Gabor David Pasztor and Marton Szemenyei. Online RPG Environment for Reinforcement Learning. In *Proceedings of the 23rd International Symposium on Measurement and Control in Robotics (ISMCR)*, pages 1–6. IEEE, 2020.
- [13] Rehman Qureshi, Cody Roberts, Manuel Indaco, Lucy Bone, Emily Kimbrell, Samuel Mulder, Daniel R Tauritz, and Davide Guzzetti. Modeling and Gamification Framework of Business Competition Between P-LEO Constellations. In *AIAA/AAS Astrodynamics Specialist Conference*, 2022.
- [14] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *The Journal of Machine Learning Research*, 22(1):12348–12355, 2021.

- [15] Corban G Rivera, Olivia Lyons, Arielle Summitt, Ayman Fatima, Ji Pak, William Shao, Robert Chalmers, Aryeh Englander, Edward W Staley, I Wang, et al. Tanksworld: a multi-agent environment for AI safety research. *arXiv preprint arXiv:2002.11174*, 2020.
- [16] S.J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.
- [17] Joseph Suarez, Yilun Du, Phillip Isola, and Igor Mordatch. Neural MMO: A massively multiagent game environment for training and evaluating intelligent agents. *arXiv preprint arXiv:1903.00784*, 2019.
- [18] J Terry, Benjamin Black, Nathaniel Grammel, Mario Jayakumar, Ananth Hari, Ryan Sullivan, Luis S Santos, Clemens Dieffendahl, Caroline Horsch, Rodrigo Perez-Vicente, et al. Pettingzoo: Gym for multi-agent reinforcement learning. *Advances in Neural Information Processing Systems*, 34:15032–15043, 2021.
- [19] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. StarCraft II: A New Challenge for Reinforcement Learning. *arXiv preprint arXiv:1708.04782*, 2017.



## Appendices

## Appendix A

### Changing the Environment

Sat-Tycoon is simply a provided demonstration of the capabilities of this framework. The framework is specifically designed with the ability to change its environment (i.e., simulation/game). This section serves as a guide to do that.

#### A.1 Changing the Game Logic

The first step in instantiating a new environment/game with the framework is to swap out the game logic. All game logic code is conveniently located in a directory titled “game\_logic”. Most of the files in the provided example directory are going to be exclusive to the Sat-Tycoon game. The major file that will very likely be important to any game is the game.py file. The game.py file is where the main game loop is defined and game logic from other files that happens on a schedule is called. For example, in Sat-Tycoon monthly updates to the population and customer count get called each month on the game loop’s clock. The other file that may be of note to other games is the player.py file, as every game will have players. The player file will likely look completely different from game to game, however, as different games will have different player data to manage and store. This file will still serve as a good blueprint for other games. The rest of the files in this directory are fairly exclusive to Sat-Tycoon, but may still be of use as examples for future developers looking to build a game for use on the framework.

## A.2 Changing the API

This next step is dependent on game logic being complete, or at least drafted. Now that the game logic is in place, there needs to be a way for players to interact with it and actually play the game. This is where the network API comes into the picture. In the directory titled “networking” there is another directory called “api”. This is where the different network APIs a game may require are stored. For Sat-Tycoon it uses a single file called “api.py”. With the exception of a couple API calls, each function in this file corresponds to an action a player client may request the server to take on their behalf. The exceptions are the “toggle\_ready” and “verify\_handshake” API calls. The toggle ready and verify handshake calls are used to make sure a client is prepared to play the game. Toggle ready is used to let players inform the client that they are ready to start playing so that the server doesn’t start the game before everyone is ready, and the verify handshake call is used by both the client and the server to make sure that they are successfully communicating.

Each API call in this file is listed in a dictionary named “responses” in the “getResponse” function. The key for each entry is a string that identifies the API call, which will need to exactly match the string sent by clients in the “action” parameter on the JSON object they send to the server. The value associated with this key generally matches that string as closely as possible, and is a function call to the corresponding function for that API call. It is important to look at and note that these functions are merely referenced here, not called. The entry matching the string from the client gets called at the bottom of the function, where a short if/else statement determines if there are included parameters that need to be included in the function call or not.

Every API call function in this file ends with a send function call from the WebSocket library, using the appropriate WebSocket for that API call. For example, “update\_players” sends a WebSocket call to every user, since all clients in Sat-Tycoon are supposed to know who is online. While “request\_budget” only sends a WebSocket call to the current user, since they were the one making the request and it wouldn’t make sense to share the results of their personal budget inquiry with everyone. The other thing that needs done in some of these function calls is to pull data from the game state. Examples of doing this can be seen in functions such as

“launch\_satellites” for a complex example, or “request\_date” for a simpler example. Sometimes in the case of Sat-Tycoon, player data needs to be sent to a client. Examples of this can be found in functions such as “request\_budget” and “request\_customers”.

For the Sat-Tycoon game specifically a single game related API call needs to happen to initialize some game data outside the api.py file. This is a call to “update\_budget”, and it happens a couple times in the “server.py” file found in the “networking” directory. These calls can be changed or removed as needed, but are a good example of how and when a developer may need to make a game related call outside of the “api.py” file.

### A.3 Other files

Sat-Tycoon requires a few other files that have been organized into the “json\_data”, “plots”, and “utility” directories. The JSON data directory contains raw data that is used by the game, and gets called by the game logic and player file as necessary. This includes things like map data, population data, and tech tree data. The utility file contains Python files with scripts that are used for non-game related utility like generating population data, converting data to geoJSON format for the React player client map, and generating track data for plotting orbit ground tracks. Other games may have similar needs, and these things serve as good examples of how to handle them.

### A.4 Developing clients

Once the game has been changed, it is important to note that the provided clients for Sat-Tycoon will no longer be compatible. New clients will need to be developed to fit with the new game. The only hard requirement for a client is that it uses the WebSocket protocol to connect and communicate with the server using the framework’s JSON API. This API is explained in more detail in Section 4.4.1. The provided clients and Sat-Tycoon game also have plenty of examples of API usage. Clients can be developed using any language compatible with WebSockets and can be anything including graphical user interfaces for players, command line only interfaces for AI, or even visualizer clients that involve little or no player interaction at all.

## Appendix B

### Getting Started

This section serves as a guide to help get started using the framework with the provided Sat-Tycoon environment as an example. It covers downloading and setting up the Python server, React client, Gym client, and how to host the Python server and React client. It also covers setting up the development environments for each. Development for these environments has only been performed and tested using a Linux environment, and this guide covers setting up such an environment as well.

#### B.1 Setting up Linux

The recommended operating system for development work on the framework is Linux<sup>1</sup>. The framework and Sat-Tycoon were specifically developed on the Linux distribution Ubuntu<sup>2</sup> version 20.04.6 LTS<sup>3</sup>.

##### B.1.1 Setting up Linux using WSL2 in Windows 10/11

The framework was developed almost exclusively using Microsoft's Windows Subsystem for Linux 2 (WSL2) environment. This is the easiest method for most people, as they already have a Windows 10 or Windows 11 machine capable of running WSL2. Official WSL documentation can be found at <https://learn.microsoft.com/en-us/windows/wsl/>, and an official installation guide can be found at <https://learn.microsoft.com/en-us/>

---

<sup>1</sup><https://www.linux.com/what-is-linux/>

<sup>2</sup><https://ubuntu.com/>

<sup>3</sup><https://www.releases.ubuntu.com/focal/>

windows/wsl/install. The Ubuntu version 20.04.6 LTS operating system can be found in the official Microsoft Store in Windows 10/11 at <https://apps.microsoft.com/store/detail/ubuntu/9PDXGNCFSCZV>. Ubuntu also has an official guide on installing their operating system on WSL at <https://ubuntu.com/tutorials/install-ubuntu-on-wsl2-on-windows-11-with-gui-support#1-overview>.

### B.1.2 Setting up a dedicated Linux install

Another option is to directly install Linux as the sole operating system on a machine. This may be a preferable option if you are setting up a dedicated hosting machine for the server or React client, or perhaps your own web-based client. It can also just be useful to have a dedicated Linux machine for development purposes. The recommended Linux distro for the framework and Sat-Tycoon is Ubuntu, as that is what both pieces of software were developed and tested on. If you plan on actively developing on the Ubuntu installation, it may be better to install the desktop version of the operating system. The official Ubuntu documentation can be found at <https://docs.ubuntu.com/> and an installation tutorial for the desktop version of the operating system can be found at <https://ubuntu.com/tutorials/install-ubuntu-desktop#1-overview>. However, for a dedicated hosting machine it may be better to use Ubuntu Server. Official Ubuntu server documentation can be found at <https://ubuntu.com/server/docs> and an official installation guide can be found at <https://ubuntu.com/server/docs/installation>.

## B.2 Using Git and GitHub

The source code for the framework and the provided Sat-Tycoon game are available on GitHub at <https://github.com/Sat-Tycoon/Satellite-Tycoon>. In order to take advantage of GitHub and download the source code, you need Git<sup>4</sup>. Git is a version control management software widely used by software developers to not only track changes but make source code public. A guide on installing Git can be found here <https://git-scm.com/>

---

<sup>4</sup><https://git-scm.com/>

book/en/v2/Getting-Started-Installing-Git. GitHub has an official quick-start guide here <https://docs.github.com/en/get-started/quickstart/hello-world> and a tutorial on Git in that same guide, located here <https://docs.github.com/en/get-started/using-git/about-git>.

### B.3 Installing the Backend

The framework's backend, or server, is located on GitHub here <https://github.com/Sat-Tycoon/Satellite-Tycoon/tree/main/backend>. It is written using Python 3, and has a couple dependencies you will need before getting started with it. Python 3 should come preinstalled with Ubuntu, however, if you've installed your own choice of Linux distribution you may need to get Python 3. Python provides documentation on that here <https://docs.python.org/3/using/unix.html>. Once you have Python 3 and you have cloned down the repository for the backend, you will need to install the other dependencies using a package manager for Python such as Python's preferred package manager, pip. Documentation on pip can be found here <https://docs.python.org/3/installing/index.html>. To get the server running, you will need to install numpy<sup>5</sup> and WebSockets<sup>6</sup>. These should both be installable with the pip install command and their name, such as "pip install websockets". However, if you encounter difficulty both have installation help on their websites in the footnotes.

### B.4 Recommended Development Environment

The author of this thesis recommends using Visual Studio Code<sup>7</sup> with WSL 2 for development on the framework and included Sat-Tycoon game. Using Visual Studio Code and WSL 2 in tandem can be a very streamlined work environment, allowing developers to use a Windows machine but have a Linux work environment. This is because Visual Studio Code has an extension allowing it to connect directly to WSL 2, and run a Linux terminal inside the editor.

---

<sup>5</sup><https://numpy.org/>

<sup>6</sup><https://pypi.org/project/websockets/>

<sup>7</sup><https://code.visualstudio.com/>

From this terminal you can run the server and clients without needing a separate window for WSL 2. A guide on setting up Visual Studio Code with WSL as a development environment can be found here <https://code.visualstudio.com/docs/remote/wsl>.

## B.5 Running the Server

Once you've installed the backend, you will want to run the server and make sure it is working. In the repository, navigate to the backend directory and then to the src directory. From here, run the command "python3 main.py", this should start the server. The server will not do much until a client has connected to it, however. It is recommended to use a client to test the server if possible.

## B.6 Installing the React Client

To install the React client, after downloading the repository navigate to the frontend directory, then the src directory from there. The React frontend is written in JavaScript, and utilizes a package manager called Yarn<sup>8</sup>. It is important to note that simply installing Yarn with the Linux distribution's package manager (Such as apt install) usually does not work, and will give the incorrect software. There are installation instructions on Yarn's website for their particular package. In the case that you experience trouble with the corepack software that Yarn recommends, you can also install Yarn directly using Node Package Manager (npm). The command to do so is "sudo npm install -g yarn". Yarn will handle installing all of the dependencies the client requires. To install these dependencies, simply run the command "yarn" from the command line. Once the dependencies are installed, the React client can be run with the "yarn dev" command.

## B.7 Installing the Gym Client

The Sat-Arena gym is hosted on GitHub alongside the framework. The first step is to clone down the Sat-Arena repo from there. Once you have it, you need to install Gymnasium<sup>9</sup>. To

---

<sup>8</sup><https://yarnpkg.com/>

<sup>9</sup><https://gymnasium.farama.org/>



do this, simply use the pip install command: “pip install gymnasium”. After Gymnasium is installed, you will need to install the actual gym environment. To do this, navigate to the sat-arena folder located inside the “gym\_envs” folder, inside the src folder. In this directory you should see a setup.py file, you don’t need to touch the file, but you need to be in the same directory as the file. From here, you need to run the command “pip install -e .”. This should detect and install the gym environment from the local setup.py file. To run the gym, navigate back to the src directory and run “python3 main.py”.

## B.8 Hosting the Server

Before connecting a client, you will need to make sure that the server is hosted and can be connected to. Start by making sure that the IP address and port are correct. To do this, navigate to the server.py file located in the networking folder found in the backend directory. In this file you should see a server class is defined. It has member variables for “production\_hostname” and “development\_hostname”. This is where the IP address for the server machine should be put. There is a production and development hostname so that a developer working on the framework can safely test their game locally using the development hostname, instead of risking changing the hostname a project may be using live in production. To run the server locally, without going out to the internet, use the localhost address “127.0.0.1”. Once you have changed the IP address, depending on which of these two environments you wish to use, you need to change the “env” variable defined in main.py in the src folder. From here, running “main.py” from the src directory is all that is required to begin hosting the server from the desired IP address.

## B.9 Hosting the React Client

The easiest way to host the React Client for testing or personal play purposes is just to run it from localhost using the “yarn dev” command, as you do not need to set up anything at all. Just make sure the IP address in the profile.js file in the login directory of the components directory is set to 127.0.0.1 and the server’s IP address is set to the same. However, if you want to host the client so that others can play using it, things get a little trickier. The official React client

is hosted using an Apache web server on a headless Ubuntu server. Ubuntu has an official guide on setting up Apache here <https://ubuntu.com/tutorials/install-and-configure-apache#1-overview>. In step three of this guide, when the html file is being created and put in “var/www/”, you should instead put the contents of the dist folder after running yarn build into there instead. You should notice this will include an html file like they make in the tutorial, as well as a few other important files the client needs.