Optimizing Resource Strategies Through Simulation and Iterative Analysis in the Halo Wars Virtual Economy

Joel Yuhas

Independent Research

November 23, 2024

A computational approach to identifying and optimizing the fastest build strategies in the Halo Wars virtual economy, focusing on resource efficiency and problem solving using Python.

1 Abstract

This project investigates the optimization of resource collection and build strategies in the real-time strategy game Halo Wars, focusing on constructing the Scarab unit as efficiently as possible. A Python-based simulator was developed to model the game's economy, incorporating randomized build order generation and validation mechanisms to explore different build combinations. Through extensive testing, the system accurately predicted resource values and identified build orders achieving resource thresholds up to 28% faster than previous manual strategies. These results validate the effectiveness of computational methods in strategy optimization, providing valuable insights for players and developers alike.

Contents

2 Introduction

Halo Wars, a real-time strategy video game released in 2009, challenges players to manage bases, armies, and resources in order to defeat their opponents. Central to the game is an intricate economy system, where players must carefully balance resource production, technological upgrades, and unit creation to gain an edge. Mastering this economy requires strategic decisions about which buildings to construct, when to upgrade, and how to sequence actions for maximum efficiency.

One of the most powerful units in Halo Wars is the Scarab—a Super Unit that requires substantial resources and a high technological level to build. Identifying the fastest way to construct a Scarab can provide players with a significant advantage, enabling them to deploy an extremely powerful unit earlier than their opponents. However, the open-ended nature of the game, combined with numerous variables like build order and resource collection rates, makes finding the optimal strategy a complex and time-consuming problem.

This project aims to address that challenge by leveraging computational simulation to analyze and optimize build orders that effect the players in-game economy. Using Python, a simulator was developed to model the game's economy with high accuracy, enabling the evaluation of various build order strategies. By iterating through randomly generated and verified build orders, the program can be used to identify the fastest sequences to achieve specific resource thresholds, such as the 3000 resources required to build a Scarab.

The primary goal of this research was to discover a build order faster than the current personal best of 430 seconds (7 minutes and 10 seconds). Additionally, the project sought to explore advanced coding techniques, create a scalable simulation framework, and validate the findings through real-world gameplay. A stretch goal included determining the fastest way to build two Scarabs, a scenario requiring even greater optimization.

Ultimately, this work offers insights not only into Halo Wars strategies but also into the broader potential of computational methods for solving optimization problems in games and beyond. The results showcase the effectiveness of combining simulation and iterative analysis to tackle complex strategic challenges, laying the groundwork for further exploration and applications.

3 Objectives

1. **Primary Goal**:
   - Use Python to discover the fastest way to build a Scarab Super Unit in Halo Wars, optimizing resource efficiency and minimizing build time.
2. **Performance Benchmark**:
   - Attempt to surpass the personal best time of 7 minutes and 10 seconds (430 seconds) to reach 3000 resources required to start construction on the Scarab.
3. **Assumptions**:
   - Simulate ideal gameplay conditions, including the specific advantages of the "Exile" map (e.g., access to free tech levels and uncontested second base slots). More information in the Assumptions section.
4. **Exploration of Stretch Goals**:
   - Investigate strategies to build two Scarabs, requiring 6800 resources and additional upgrades, under similar optimization constraints.
5. **Skill Development**:
   - Utilize the project as a learning opportunity to practice advanced software development techniques and problem-solving in Python.
6. **Validation**:
   - Test and validate the simulator's results against actual in-game performance to ensure accuracy and practical applicability.

4 Problem Definition

4.1 Game Background

A "Scarab" is the most powerful Super Unit in the game, and also the most expensive. The motivation for finding an answer to the primary objective above is that if a player can produce the strongest unit as fast as possible, it can provide them a massive advantage against their opponents. To optimize the build process for the Scarab, it's essential to first define its specific prerequisites and the game mechanics governing its production.

Halo wars has two factions the player can play as, the "UNSC" and the "Covenant", each with their own unique abilities and special units. The Scarab is only available to the Covenant faction, and thus the player must play as the Covenant.

To build a Scarab, a player must produce enough resources (3000) and acquire a high enough Technology level (Tech level of 3). The player gains resources by creating and upgrading buildings called "Supply Pads" on their base. These buildings and upgrades cost resources to construct, but provide the player with a steady stream of resources over time after they're built.

The player also has a "Technology" level, which for the Covenant ranges from zero to three. The Technology or "Tech" level can be increased from zero to one by creating a building called a "Temple", and can be further increased by upgrading the Temple.

These are the three main requirements for building a Scarab, and they have been listed out in the next section.

4.2 Scarab Building Requirements

1. The player must play as the "Covenant" Faction.
2. The player must have 3000 resources ready to purchase the Scarab unit.
3. The player must have a current Technology level of three at the time of building the Scarab.

4.3 Game Mechanics

4.3.1 Player Abilities

Now that the core Scarab requirements have been established, understanding what actions are available to the player can show how to achieve them. Since Halo Wars is a dynamic and rather open ended game, there are many actions and abilities the player can perform. However, since this project is only exploring the fastest build time of a specific unit, many abilities and actions can be safely ignored for now. Below is a high-level overview of the core actions available to a player in Halo Wars:

- Build a Base.
- Build a Building on one of the Base's Build Slots.
- Upgrade a Building or Base.
- Create a Unit.

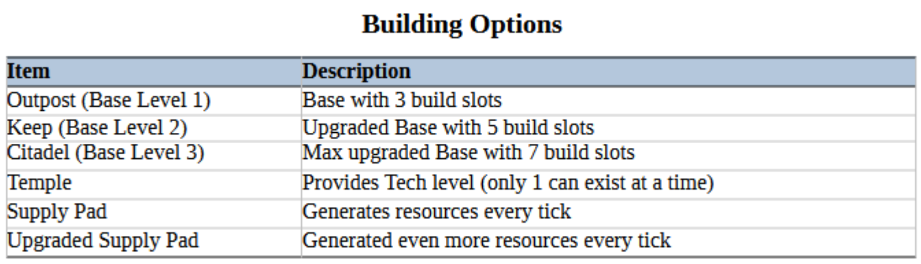
Bases allow the player access up to seven "Build Slots" and can build specific units (including the Scarab). Each player always starts with one base, and other bases can be constructed on different "Base Slots" available on the map.

Build Slots are connected to bases and are where "buildings" can be constructed. Build Slots start off empty and the player can choose from several building types that can be constructed on that slot. There are either three, five, or seven build slots on a base depending on what level the base is.

Buildings are built on build slots and provide the player some utility. For this project, the only buildings that are considered are buildings that increase the players tech level (Temple) or supply continuous resources (Supply Pad).

Units are troops that can move across the map. They can be used to attack other players and perform special actions. For this project, the only major units that needs to be considered are the Scarab, the Covenant Leader unit that is supplied after building a Temple, and a basic infantry unit used later on.

With the basics of Bases, Build Slots, Buildings, and Units explained, a more refined list of actions available to the player can be determined. This will be useful in figuring out what actions to account for when developing the simulator later on. Figure 1 shows a more detailed list of the Base levels, the buildings and their upgrades:



Building Options

Item	Description
Outpost (Base Level 1)	Base with 3 build slots
Keep (Base Level 2)	Upgraded Base with 5 build slots
Citadel (Base Level 3)	Max upgraded Base with 7 build slots
Temple	Provides Tech level (only 1 can exist at a time)
Supply Pad	Generates resources every tick
Upgraded Supply Pad	Generated even more resources every tick

Figure 1: Building and base options available to the player that affect build slot amount, economy, and tech level.

4.3.2 Action Requirements

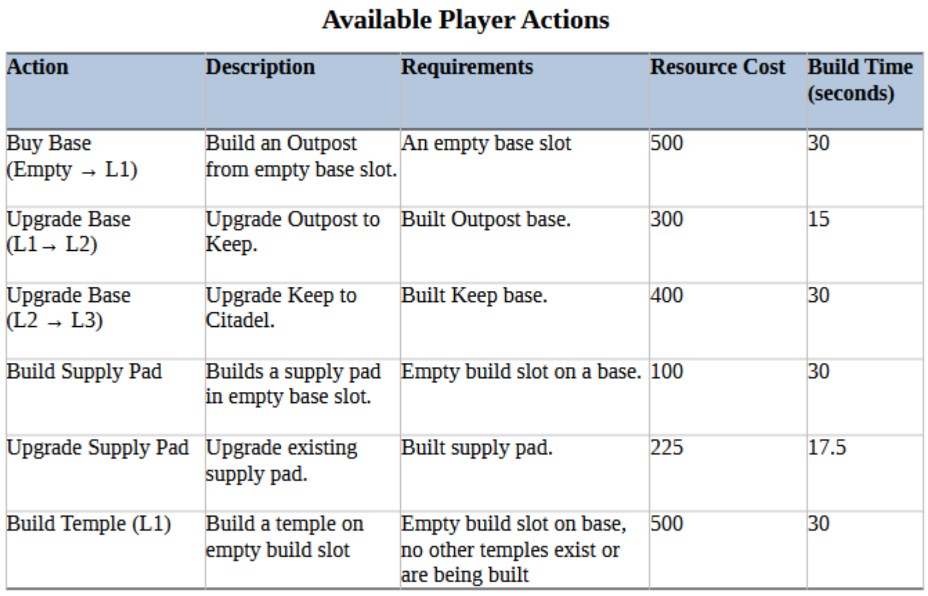
Each action and ability offers a benefit at the expense of costing resources to construct. Each ability also does not happen instantly and will take a specific amount of time for that respective ability to be performed. The table in Figure 2 contains all the relevant actions a player can do along with their respective resource and time cost.

In-game, if a player attempts an action but doesn't have the requirements (e.g. does not have enough resources), no penalizing action is taken, the action is simply not completed. Additionally, in-game, once a building has been built on a build slot, there normally is an option to recycle it, which destroys the building and frees the build slot again. However, for simplicity, this project will not have the recycle option, and once a building is built it will stay in that build slot indefinitely.

4.3.3 Starting Scenario

With the primary actions explained, it is now important to understand how each Halo Wars game starts. When a match starts, the player begins with the following:

- A level two Base with five open build slots
- A Tech level of zero
- 800 starting resources.



Available Player Actions

Action	Description	Requirements	Resource Cost	Build Time (seconds)
Buy Base (Empty → L1)	Build an Outpost from empty base slot.	An empty base slot	500	30
Upgrade Base (L1→ L2)	Upgrade Outpost to Keep.	Built Outpost base.	300	15
Upgrade Base (L2 → L3)	Upgrade Keep to Citadel.	Built Keep base.	400	30
Build Supply Pad	Builds a supply pad in empty base slot.	Empty build slot on a base.	100	30
Upgrade Supply Pad	Upgrade existing supply pad.	Built supply pad.	225	17.5
Build Temple (L1)	Build a temple on empty build slot	Empty build slot on base, no other temples exist or are being built	500	30

Figure 2: Actions along with their resource and time cost.

The player also begins with a free "scout" unit that has basic offensive capabilities and can explore the map. There also are multiple batches of "supplies" on the map, each batch can give the player between 100-200 extra resources. The player must use unit to collect these supplies. For the sake of this project, the supplies gathered by units on the map were not calculated, since the results in-game are often inconsistent. Instead, there were some test scenarios devised to simulate resource gathering by changing the target resource threshold.

Additionally, when a player builds their Temple for the first time, they are gifted a special "Leader Unit". This unit has offensive capabilities that will be utilized in-game, but are not necessary to note in the economy simulation.

4.3.4 Ideal/Project Scenario

Halo wars has several "maps" that can be played, each of which offer special perks and abilities that are not otherwise available on other maps. For this project, there is one map in particular that can expedite the creation of a Scarab called "Exile". What makes Exile special is there are four "Reactors" guarded by third party hostile enemies. If the player garrisons an infantry unit they control in a reactor, their tech level increases by one as long as they hold it. It should be noted that the scout and leader units can NOT garrison the Reactors, and it needs to be done by a separate "infantry" unit.

This means if the player gets two of these reactors and builds a Temple, they will reach a tech level of three without needing to spend time and resources upgrading their Temple. This drastically speeds up the Scarab requirements completion process and over halves the resource requirements (upgrading the Temple twice cost 3000 total resources). In-game, these reactors can be cleared of hostile mobs by the players Leader Unit, as well as with help from their scout unit. In the simulation, this is assumed to happen successfully and no further calculations are required.

Additionally, to get a second base on a traditional Halo Wars map, the player has to clear out a Base Slot that is occupied by other hostile third party enemies. However, on the map Exile, there are several free Base Slots already available that can be taken with no contest. This means a second base is available to the player almost immediately, instead of normally taking two to three minutes to clear out.

Because of the advantages that the map Exile offers, the simulation assumes it will automatically have access to the Reactors and uncontested base slots. All major other assumptions are further detailed in the next section.

4.4 Assumptions

After understanding the major game mechanics, there are a few areas that may add either confusion or complexity. To help simplify things and create a consistent foundation for the project, the following are the major assumptions that are being accounted for.

- While a Supply Pad is being built, it does NOT produce resources.
- While a Supply Pad is being upgrade, it will continue to count as an un-upgraded Supply Pad until the moment the upgrade is complete.
- The cost for the two extra infantry units needed for garrisoning in generators can either be added to the final resources threshold (3300), or can be ignored with the assumption that those resources would be gathered in game from supplies on the map (3000).
- Adding the two extra infantry units will NOT add time complexity.
- Assume the Reactors are cleared and populated by infantry units in time.
- For the the simulator, the only resources that will be added to the resource count are the resources from upgraded/non-upgraded Supply Pads, and the 800 starting resources.
- If going for the two Scarab stretch goal, the "Reinforcements" upgrade is required, which costs 800 additional resources (6800 total).

4.5 Game Economy

4.5.1 Original Equation

Now that the player actions and project scenario has been established, a deeper look at the Halo Wars in-game economy is crucial to understating how to simulate it.

Halo Wars uses a specialized formula that relies on how many Supply Pads and Upgraded Supply Pads a player has. Because it follows a non-linear formula, and because the variables in the formula (number and type of Supply Pads) changes over time, it makes it difficult to accurately calculate how many resources a player should have as the game progresses. The original equation for how the amount of resources are calculated was found on an online discussion form and is presented in Equation 1.

$$S = \frac{1.5X}{\left(\frac{X+Y}{13}\right)+1} + \frac{2.5Y}{\left(\frac{X+Y}{13}\right)+1} \tag{1}$$

- S: Supplies per second
- X: Quantity of upgraded Supply Pads
- Y: Quantity of regular Supply Pads

4.5.2 Modified Equation

Initial prototype code was created and ran using this equation. However, when comparing simulation results with in game testing, it was found that this equation was slightly off, as the expected resources for even basic tests would deviate from the actual results after only a few minutes. After repeated experimentation and close analysis of expected vs actual results in a variety of situations, it was found that the following, updated equation show in Equation 2 produced significantly more accurate results.

$$S = \frac{4.375X}{\left(\frac{X+Y}{9}\right)+1} + \frac{6.125Y}{\left(\frac{X+Y}{9}\right)+1} \tag{2}$$

The main changes being adding 1 to the X and Y coefficients and further multiplying them by another 1.75, as well as reducing the denominator from 13 to 9. Needing to modify the original equation was not unexpected, as several threads on the topic noted that the equation seemed to be from an early version of the game that had since been modified. It is likely that the actual in-game equation is different as well, as the resource value is updated every frame (60 frames a second), where as this equation calculates resources every second.

4.5.3 Equation Visualizations

Several graphs were made to help visualize the relationship between upgraded or "heavy" Supply Pads,"lite" or regular Supply Pads, and how many resources they together produce a second. Figure 3 shows the direct impact of heavy(Y) and lite(X) Supply Pads on the supplies per second output (S).

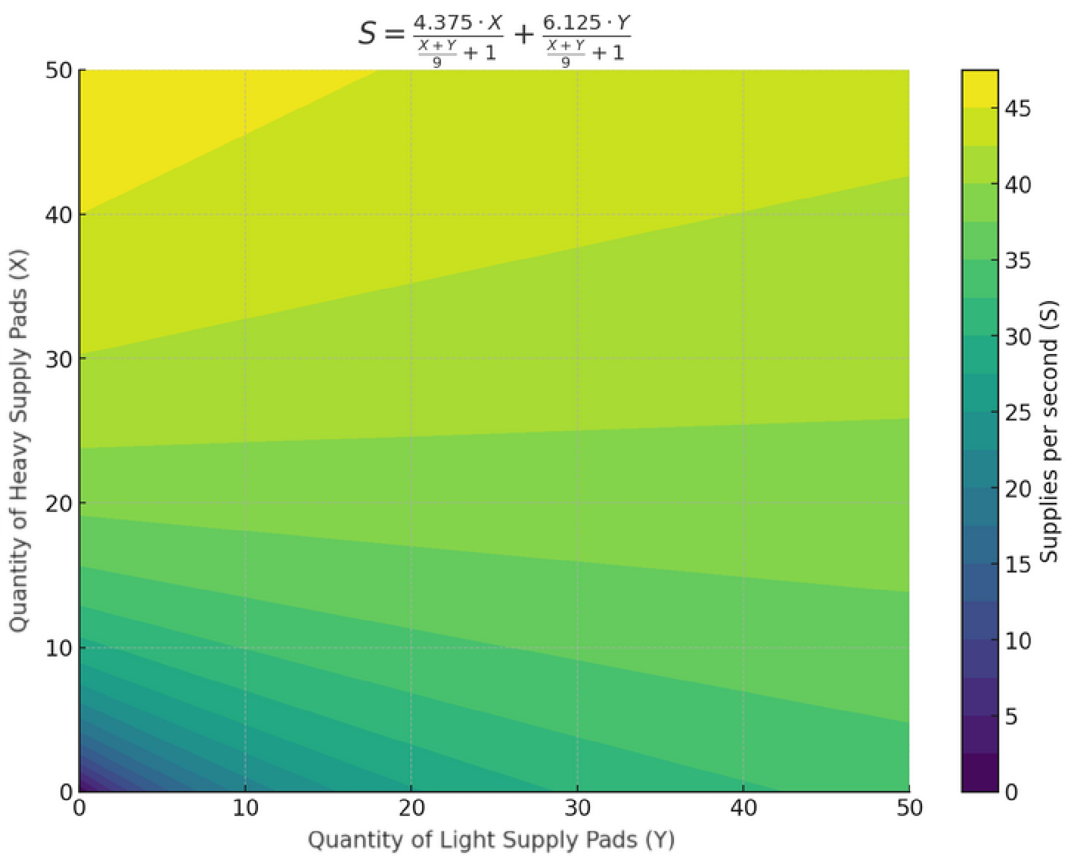


Figure 3: Supply Equation Visualization.

However, Figure 3 only shows the direct impact that X and Y have on S, it does not take into the account the impact X and Y have on each other. The following is a more detailed breakdown for constructing a more truthful visualization.

- Since X and Y relate to Supply Pads that are either built or not built, that means X and Y can only be integer values.
- Since upgraded Supply Pads can only come from un-upgraded supply Pads, Y can only increase when X is greater than 0.
- Additionally, when Y increases, X decreases by the same amount

Figure 4 shows a more detailed visualization taking into account the criteria above.

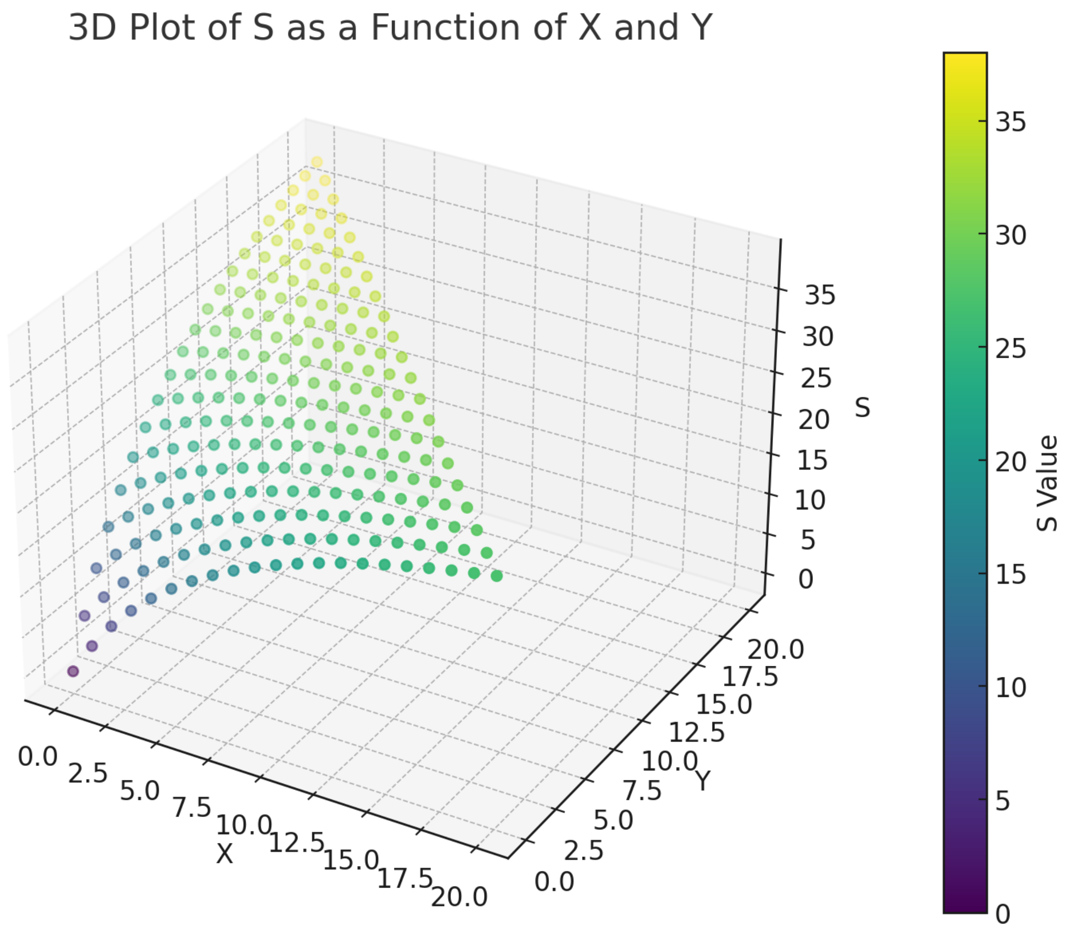


Figure 4: Detailed Supply Equation Visualization.

Finally, Supply Pads do not just produce resources, they also cost resources to construct. The impact of X and Y on how many resources they cost can be visualized as well. The resource cost can be dictated as follows:

- Increasing X by one cost 100 resources.
- Increasing Y by one cost 225 resources.

Figure 5 gives a visualization of the cumulative cost when purchasing supply pads for any given combination up to twenty.

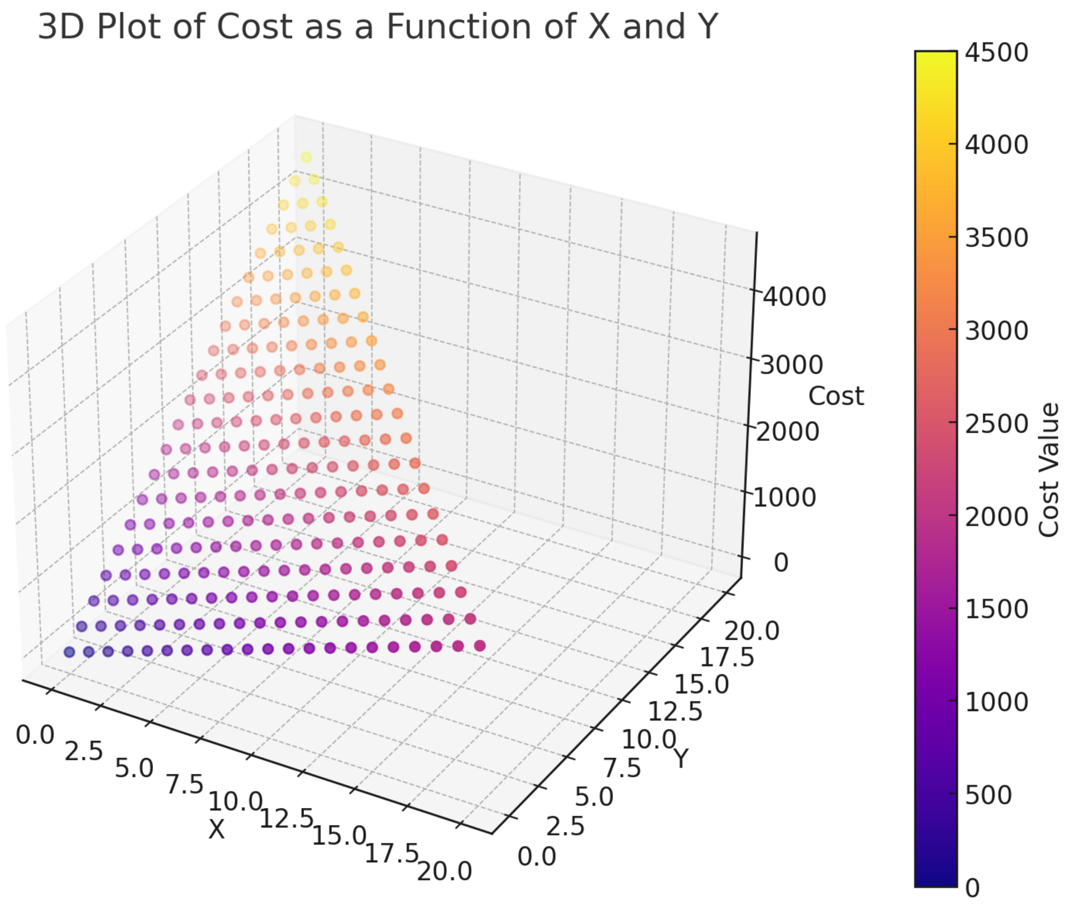


Figure 5: Visualization of Lite and Heavy Supply Pad Costs.

5 Equipment

- **Operating System**: Windows 10
- **Development Environment**: Pycharm
- **Programming Language**: Python 3.11.1
- **Testing Platform**: Halo Wars Definitive Edition (Xbox One)

6 Methodology

6.1 Introduction

This section outlines the methodology used to develop the Halo Wars economy simulator and build order optimizer. The approach consisted of designing a Python-based simulator, developing a way to test different build orders, and combining these components to identify the fastest Scarab build strategy.

6.2 Overall Design

The simulator aimed to mimic the in-game mechanics of Halo Wars. Those mechanics, described in the previous sections, primarily involve resource generation calculations, building construction timing, and upgrade timing. To achieve this, Python was used to create this simulator, utilizing object oriented programming and software design principals.

Several prototypes were explored, but the final result involved having the simulator take in a "build order"—a list of sequential actions to execute—and calculate the time required to reach a specific resource threshold. To identify the fastest build order, a secondary program was created to generate random build orders for the simulator to evaluate. The idea behind random build orders was that if enough random orders were generated, the fastest build order, or at least certain trends, could be found.

The project was divided into two main components:

- **Simulator program:** Kept track of build times and the resource collection rates, outputting the time required to reach a given resource threshold in in-game seconds.
- **Build order generator:** Generated randomized build orders to feed into the simulator for evaluation. Certain redundancy and verification checks were utilized to provide only valid build orders to the simulator.

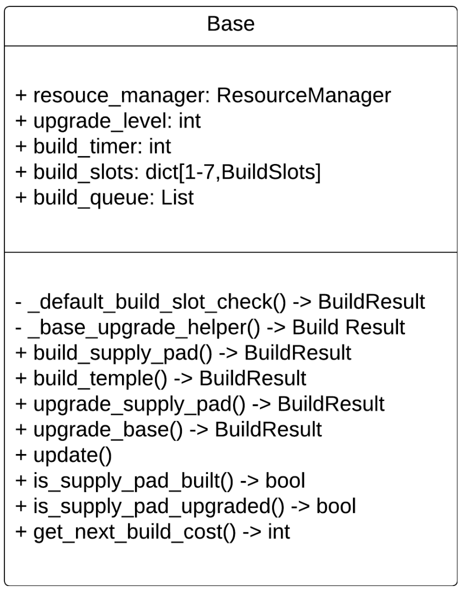
Once developed, the two components were integrated using a simulator wrapper and supporting classes to streamline execution and analysis.

6.3 Python Simulation Components Design

The simulator was designed with a modular architecture consisting of three primary components:

- **Base class:** Managed base operations, including construction, upgrades, and build timers.
- **BuildSlot class:** Represented individual build slots for constructing and upgrading buildings.
- **ResourceManager class:** Oversaw resource and technology level updates to ensure consistency.

6.3.1 Base Class



```
Base

+ resouce_manager: ResourceManager
+ upgrade_level: int
+ build_timer: int
+ build_slots: dict[1-7,BuildSlots]
+ build_queue: List

- _default_build_slot_check() -> BuildResult
- _base_upgrade_helper() -> Build Result
+ build_supply_pad() -> BuildResult
+ build_temple() -> BuildResult
+ upgrade_supply_pad() -> BuildResult
+ upgrade_base() -> BuildResult
+ update()
+ is_supply_pad_built() -> bool
+ is_supply_pad_upgraded() -> bool
+ get_next_build_cost() -> int
```

Figure 6: Base Class

The **Base** class modeled individual bases in the game. Each base has up to seven build slots and operates independently, with its own build timers and slot assignments. This independence ensures accurate simulation of multi-base gameplay, where actions on one base do not interfere with others.

Key methods include:

- `build_supply_pad()` and `build_Temple()`: For constructing Supply Pads and Temples.
- `upgrade_base()` and `upgrade_supply_pad()`: For upgrading bases and Supply Pads.
- `update()`: Called every in-game second to manage build timers and update associated build slots.

Additionally, the class implemented a **BuildResult** enumeration to provide detailed feedback on action outcomes (e.g., `NOT_ENOUGH_RESOURCES`, `APPROVED`). A **BaseState** enumeration tracked the base's status (e.g., `IDLE`, `UPGRADING`).

6.3.2 BuildSlot Class

The **BuildSlot** class served as an abstract foundation for all building types. Subclasses included:

- **EmptySlot:** Default state of unused build slots, facilitating checks for available slots.
- **SupplyPad:** Handled Supply Pad construction, upgrades, and interactions with the ResourceManager.

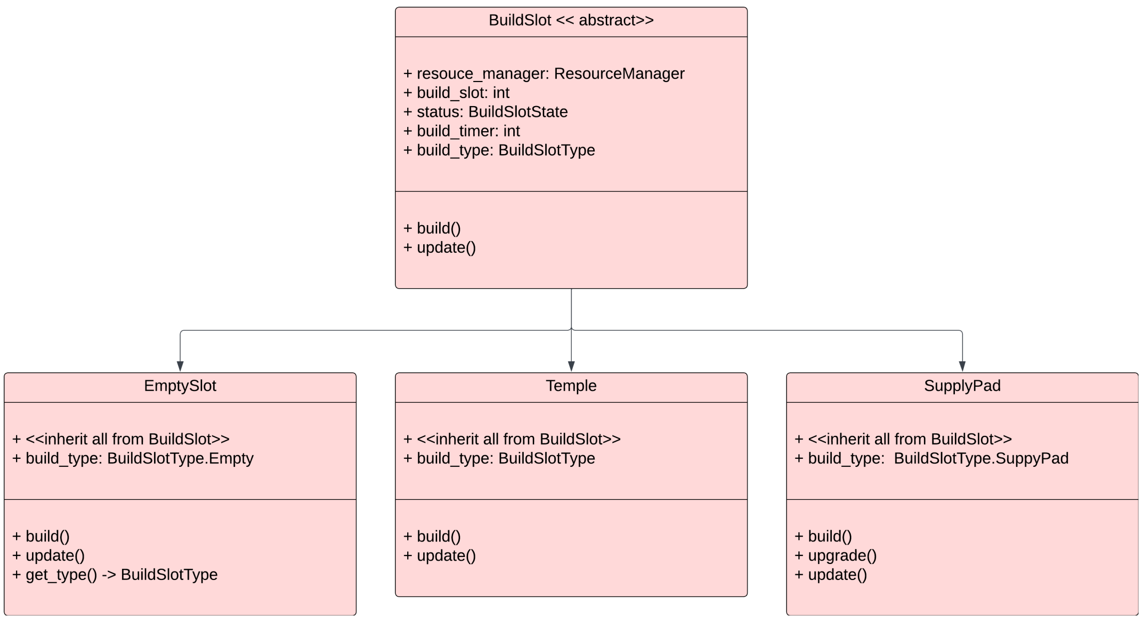

BuildSlot << abstract>>
+ resouce_manager: ResourceManager
+ build_slot: int
+ status: BuildSlotState
+ build_timer: int
+ build_type: BuildSlotType
+ build()
+ update()
EmptySlot
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType.Empty
+ build()
+ update()
+ get_type() -> BuildSlotType
Temple
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType
+ build()
+ update()
SupplyPad
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType.SuppyPad
+ build()
+ upgrade()
+ update()

Figure 7: BuildSlot Class

- **Temple:** Managed Temple construction and tech level updates.

By centralizing resource management within build slot classes, the design minimized the risk of errors, such as duplicate or untracked buildings. Constants for build costs and times were stored in a separate configuration file for easy modification.

6.3.3 ResourceManager Class

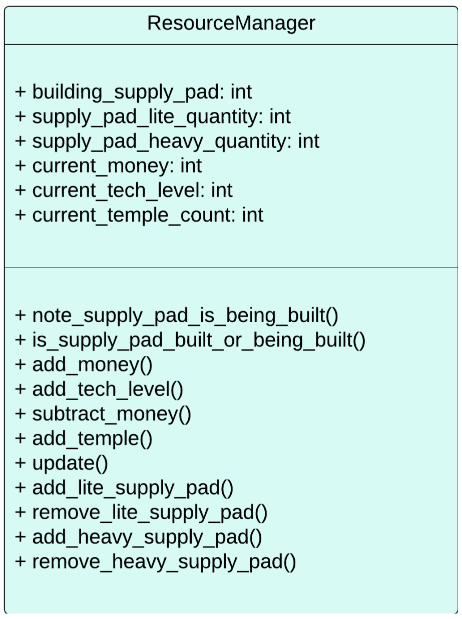

ResourceManager
+ building_supply_pad: int
+ supply_pad_lite_quantity: int
+ supply_pad_heavy_quantity: int
+ current_money: int
+ current_tech_level: int
+ current_temple_count: int
+ note_supply_pad_is_being_built()
+ is_supply_pad_built_or_being_built()
+ add_money()
+ add_tech_level()
+ subtract_money()
+ add_temple()
+ update()
+ add_lite_supply_pad()
+ remove_lite_supply_pad()
+ add_heavy_supply_pad()
+ remove_heavy_supply_pad()

Figure 8: ResourceManager Class

The **ResourceManager** class centralized resource and technology level tracking, ensuring consistency across the simulation.

Key variables included:

- `current_money`: Tracked the player's resource balance.
- `current_tech_level`: Tracked the player's technology level.
- `supply_pad_lite_quantity` and `supply_pad_heavy_quantity`: Count regular and upgraded Supply Pads.
- `building_supply_pad`: Tracked whether a Supply Pad was under construction.

Core methods:

- `add_money()` and `subtract_money()`: Managed resource transactions.
- `add_tech_level()`: Updated the tech level, allowing for future flexibility beyond the current use case.
- `update()`: Calculated resource generation each in-game second using the Supply Pad formula.

6.3.4 Combining the Simulator Classes

With all core components now developed, a working program existed that could accurately predict the Halo Wars economy when the Objects were manually controlled. Figure 9 shows the relationship between these classes and how they operated together.

However, another class was needed that could take in the build order list and turn it into actionable commands the simulator could execute on. For that, the **RuntimeBuildingBlocks** class was created to coordinate their interactions. This class included:

- `run_simulator()`: Executed build orders and calculated the time required to reach a resource threshold.
- `build_verifier()`: Validated and executed individual build actions.

The `run_simulator()` method output the time taken to reach a resource threshold in integer format. The output time and the respective build order details are used in the eventual wrapper class to add the results to the final results list, and eventual CSV file. Figure 10 illustrates the relationships between the components, specifically how the **RuntimeBuildingBlocks** takes in the build order list, Base references, and resource trigger limit, and can feed it properly to the Simulator.

Note, the `run_simulator()` method was designed to attempt to run every build order. If there was an order to perform a build or upgrade, but there was not enough resources, the simulator was programmed to wait until enough resources are generated before executing and proceeding.

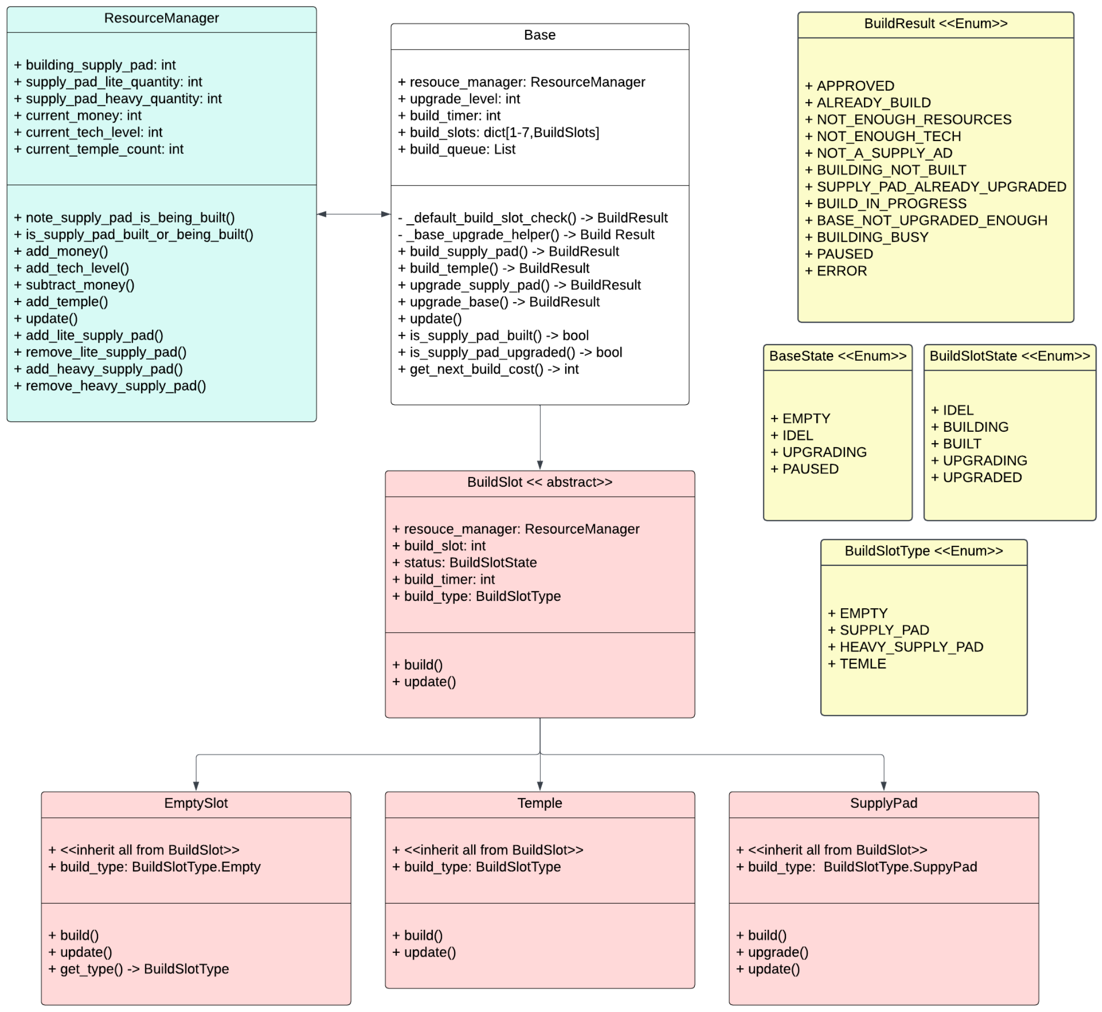
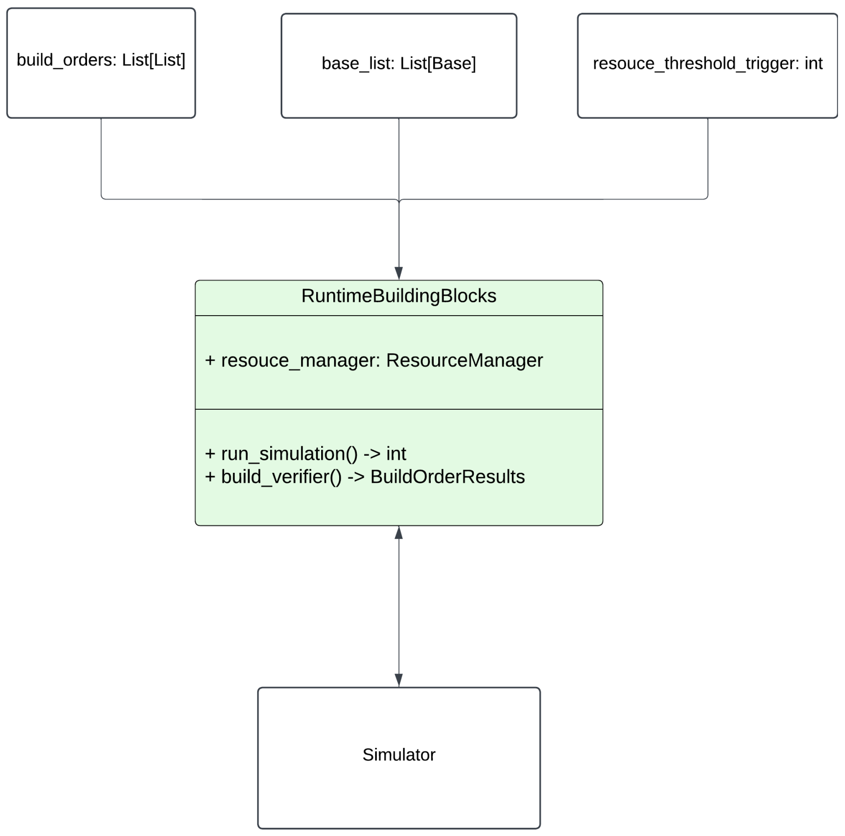

ResourceManager
+ building_supply_pad: int
+ supply_pad_lite_quantity: int
+ supply_pad_heavy_quantity: int
+ current_money: int
+ current_tech_level: int
+ current_temple_count: int
+ note_supply_pad_is_being_built()
+ is_supply_pad_built_or_being_built()
+ add_money()
+ add_tech_level()
+ subtract_money()
+ add_temple()
+ update()
+ add_lite_supply_pad()
+ remove_lite_supply_pad()
+ add_heavy_supply_pad()
+ remove_heavy_supply_pad()
Base
+ resouce_manager: ResourceManager
+ upgrade_level: int
+ build_timer: int
+ build_slots: dict[1-7,BuildSlots]
+ build_queue: List
- _default_build_slot_check() -> BuildResult
- _base_upgrade_helper() -> Build Result
+ build_supply_pad() -> BuildResult
+ build_temple() -> BuildResult
+ upgrade_supply_pad() -> BuildResult
+ upgrade_base() -> BuildResult
+ update()
+ is_supply_pad_built() -> bool
+ is_supply_pad_upgraded() -> bool
+ get_next_build_cost() -> int
BuildResult <<Enum>>
+ APPROVED
+ ALREADY_BUILD
+ NOT_ENOUGH_RESOURCES
+ NOT_ENOUGH_TECH
+ NOT_A_SUPPLY_AD
+ BUILDING_NOT_BUILT
+ SUPPLY_PAD_ALREADY_UPGRADED
+ BUILD_IN_PROGRESS
+ BASE_NOT_UPGRADED_ENOUGH
+ BUILDING_BUSY
+ PAUSED
+ ERROR
BaseState <<Enum>>
+ EMPTY
+ IDEL
+ UPGRADING
+ PAUSED
BuildSlotState <<Enum>>
+ IDEL
+ BUILDING
+ BUILT
+ UPGRADING
+ UPGRADED
BuildSlot << abstract>>
+ resouce_manager: ResourceManager
+ build_slot: int
+ status: BuildSlotState
+ build_timer: int
+ build_type: BuildSlotType
+ build()
+ update()
BuildSlotType <<Enum>>
+ EMPTY
+ SUPPLY_PAD
+ HEAVY_SUPPLY_PAD
+ TEMLE
EmptySlot
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType.Empty
+ build()
+ update()
+ get_type() -> BuildSlotType
Temple
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType
+ build()
+ update()
SupplyPad
+ <<inherit all from BuildSlot>>
+ build_type: BuildSlotType.SuppyPad
+ build()
+ upgrade()
+ update()

Figure 9: Basic Halo Wars Simulator UML Diagram

6.4 Python Build Order Program Design

6.4.1 Build Order List Structure

The concept of "Build Orders" was central to the program's functionality. After experimentation, the final structure consisted of creating a list of Build Orders that could be read by the program. The orders in the list would give information for performing specific actions that the program would try to execute. Each order consisted of three key components:

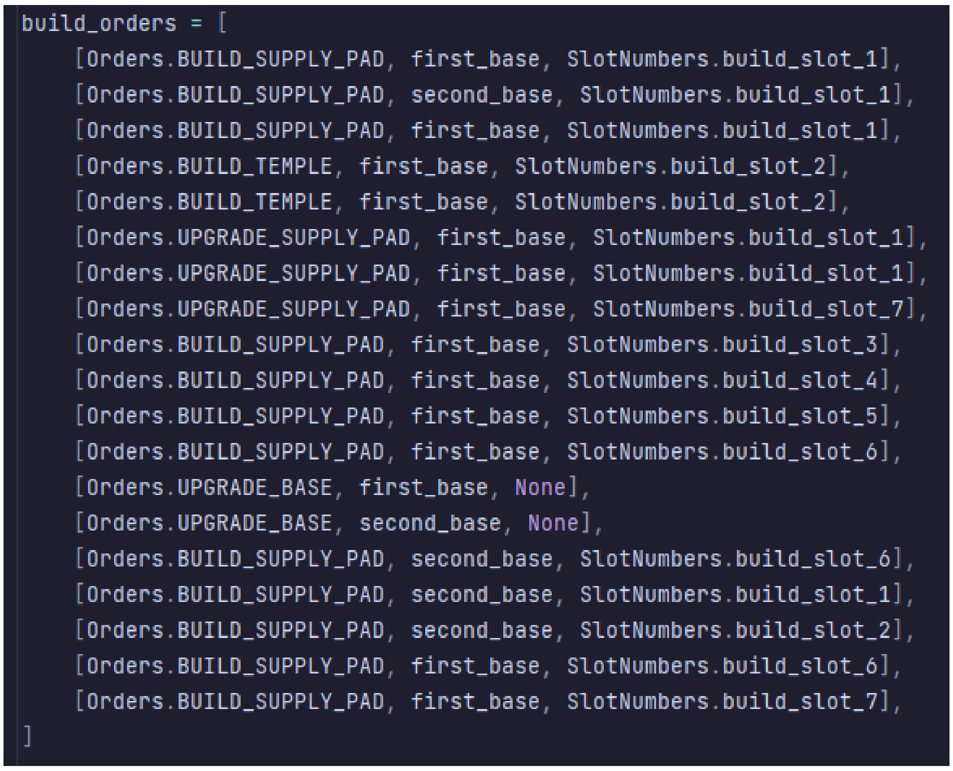


build_orders: List[List]
base_list: List[Base]
resouce_threshold_trigger: int
RuntimeBuildingBlocks
+ resouce_manager: ResourceManager
+ run_simulation() -> int
+ build_verifier() -> BuildOrderResults
Simulator

Figure 10: RuntimeBuildingBlocks Relationship

- **Order Enumeration:** Specifies the action to perform, including:
  - `BUILD_SUPPLY_PAD`
  - `BUILD_TEMPLE`
  - `UPGRADE_BASE`
  - `UPGRADE_SUPPLY_PAD`
- **Base Number:** An integer reference to the base where the order is executed.
- **BaseSlotNumber:** Indicates the build slot for building-related orders. For base-related actions (e.g., upgrades), this field was not used.

These values were stored in individual lists, which were then combined into a larger `BuildOrderList` representing the full sequence of actions. This design allowed the simulator's `RuntimeBuildingBlocks` class to iterate through the build order, execute each step, and calculate the time required to reach a specific resource threshold. Figure 11 illustrates an example build order generated in Python.



```
build_orders = [
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_1],
    [Orders.BUILD_SUPPLY_PAD, second_base, SlotNumbers.build_slot_1],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_1],
    [Orders.BUILD_TEMPLE, first_base, SlotNumbers.build_slot_2],
    [Orders.BUILD_TEMPLE, first_base, SlotNumbers.build_slot_2],
    [Orders.UPGRADE_SUPPLY_PAD, first_base, SlotNumbers.build_slot_1],
    [Orders.UPGRADE_SUPPLY_PAD, first_base, SlotNumbers.build_slot_1],
    [Orders.UPGRADE_SUPPLY_PAD, first_base, SlotNumbers.build_slot_7],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_3],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_4],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_5],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_6],
    [Orders.UPGRADE_BASE, first_base, None],
    [Orders.UPGRADE_BASE, second_base, None],
    [Orders.BUILD_SUPPLY_PAD, second_base, SlotNumbers.build_slot_6],
    [Orders.BUILD_SUPPLY_PAD, second_base, SlotNumbers.build_slot_1],
    [Orders.BUILD_SUPPLY_PAD, second_base, SlotNumbers.build_slot_2],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_6],
    [Orders.BUILD_SUPPLY_PAD, first_base, SlotNumbers.build_slot_7],
]
```

Figure 11: Example Build Order List

6.4.2 Generating Build Orders

The original, most pressing question of this project was how to find the optimal build order. Initially, statistical methods were explored to determine every possible combination of build orders. However, that proved to be too difficult, so the idea of generating random build orders was born.

A program was developed to generate these random build orders and specific safety checks would ensure they were non-repeating and valid. The main caveat with this method was that sufficient iteration coverage was needed to increase the likelihood of finding the optimal solutions.

The core methodology was implemented in the `GenerateOrdersBuildingBlocks` class and followed these steps:

- A **maximum build order length** was defined (default: 28).
- A random number determined the length of the build order that was to be generated (how many total orders).
- A list of currently built bases was tracked, and a random base was selected.
- A random action was chosen based on predefined probabilities:
  - Actions such as `BUILD_SUPPLY_PAD` or `UPGRADE_SUPPLY_PAD` had higher probabilities due to their frequent use.
  - Unique actions like building a `Temple` were assigned lower probabilities.

- These probabilities were stored as adjustable parameters to allow easy tuning during testing and development.

This randomization approach efficiently generated diverse build orders. Figure 12 provides an overview of the `GenerateOrdersBuildingBlocks` class.

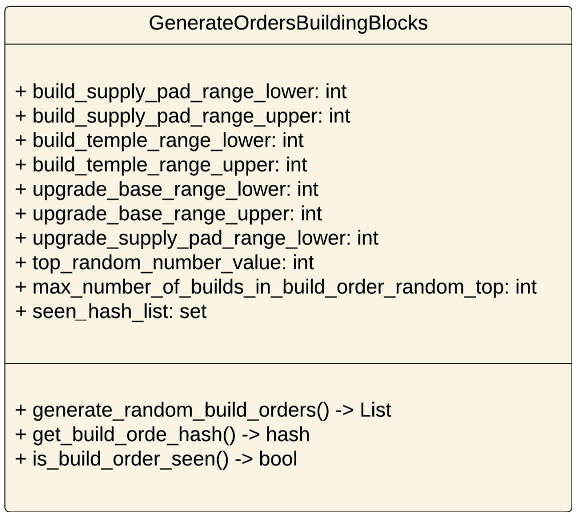

GenerateOrdersBuildingBlocks
+ build_supply_pad_range_lower: int
+ build_supply_pad_range_upper: int
+ build_temple_range_lower: int
+ build_temple_range_upper: int
+ upgrade_base_range_lower: int
+ upgrade_base_range_upper: int
+ upgrade_supply_pad_range_lower: int
+ top_random_number_value: int
+ max_number_of_builds_in_build_order_random_top: int
+ seen_hash_list: set
+ generate_random_build_orders() -> List
+ get_build_orde_hash() -> hash
+ is_build_order_seen() -> bool

Figure 12: GenerateOrdersBuildingBlocks Class

6.4.3 Build Order Safety Mechanisms

To ensure valid and efficient build orders, several safeguards were implemented:

- The `BaseBuildCounters` class tracked key metrics such as the number of bases, their upgrade levels, available slots, and built structures.
- Build orders were validated during generation to prevent invalid actions (e.g., exceeding the number of available slots for Supply Pads).
- Helper variables, such as `build_index`, were used to sequentially assign actions to appropriate slots, minimizing randomness in slot allocation.
- A flag was made that could be toggled to ensure a Temple was in the build order (when first testing, none of the fastest build orders contained temples, which was invalid).

For example, the `build_index` ensured that Supply Pads are assigned to available slots in order, while a separate `temple_index` tracked the Temple slot to avoid conflicts with Supply Pad upgrades. Figure 13 illustrates the `BaseBuildCounters` class.

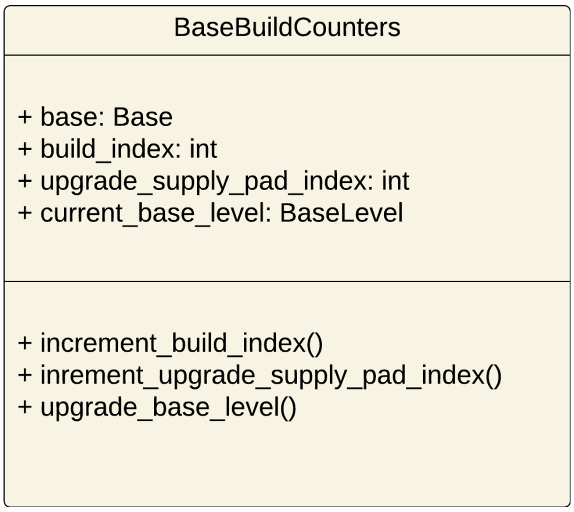

BaseBuildCounters
+ base: Base
+ build_index: int
+ upgrade_supply_pad_index: int
+ current_base_level: BaseLevel
+ increment_build_index()
+ inrement_upgrade_supply_pad_index()
+ upgrade_base_level()

Figure 13: BaseBuildCounters Class

6.4.4 Checking for Redundancies

To avoid duplicate build orders, a hash-based system was implemented:

- Each generated build order was hashed and checked against a set called `seen_hash_list`.
- If the hash was already present, the build order was discarded; otherwise, it was added to the output list.

This approach reduced computational overhead compared to iterative comparisons. Since this method stored the hashes in a set, it had a lookup complexity of $O(1)$ instead of $O(n)$ if the program were to attempted to find duplicates by looking through the build order list directly. This also highlights the use of efficient data structures such as hash sets, aligning with the project's focus on leveraging Python's strengths. Figure 14 shows the overall build generation and validation workflow.

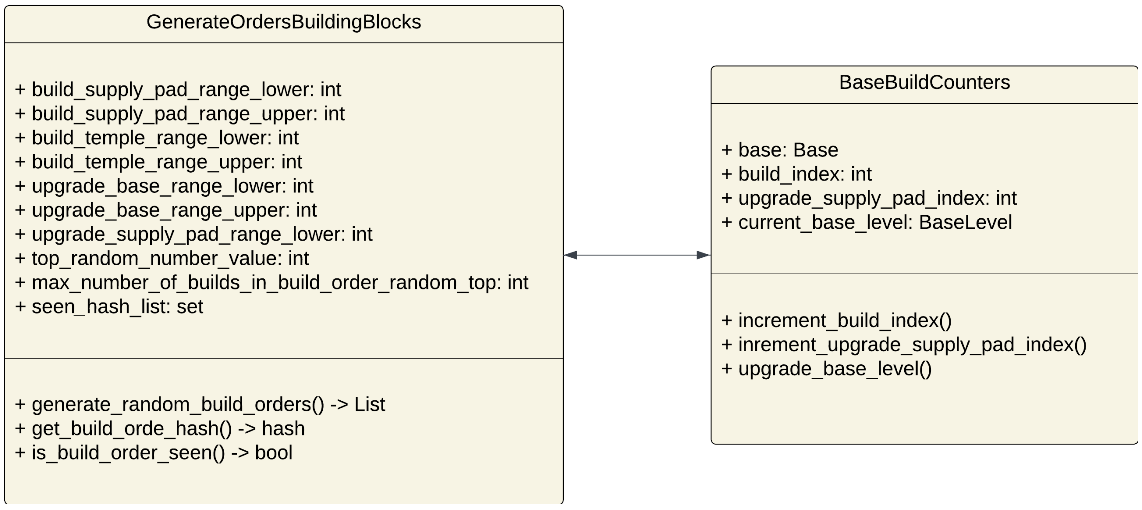

GenerateOrdersBuildingBlocks
+ build_supply_pad_range_lower: int
+ build_supply_pad_range_upper: int
+ build_temple_range_lower: int
+ build_temple_range_upper: int
+ upgrade_base_range_lower: int
+ upgrade_base_range_upper: int
+ upgrade_supply_pad_range_lower: int
+ top_random_number_value: int
+ max_number_of_builds_in_build_order_random_top: int
+ seen_hash_list: set
+ generate_random_build_orders() -> List
+ get_build_orde_hash() -> hash
+ is_build_order_seen() -> bool
BaseBuildCounters
+ base: Base
+ build_index: int
+ upgrade_supply_pad_index: int
+ current_base_level: BaseLevel
+ increment_build_index()
+ inrement_upgrade_supply_pad_index()
+ upgrade_base_level()

Figure 14: Build Generation Overview

6.5 Merging the Two Products

With the primary simulator classes, build order generation mechanisms, and validation systems complete, the project was ready to integrate all components into a cohesive workflow. Figure 15 illustrates the high-level relationships between these systems.

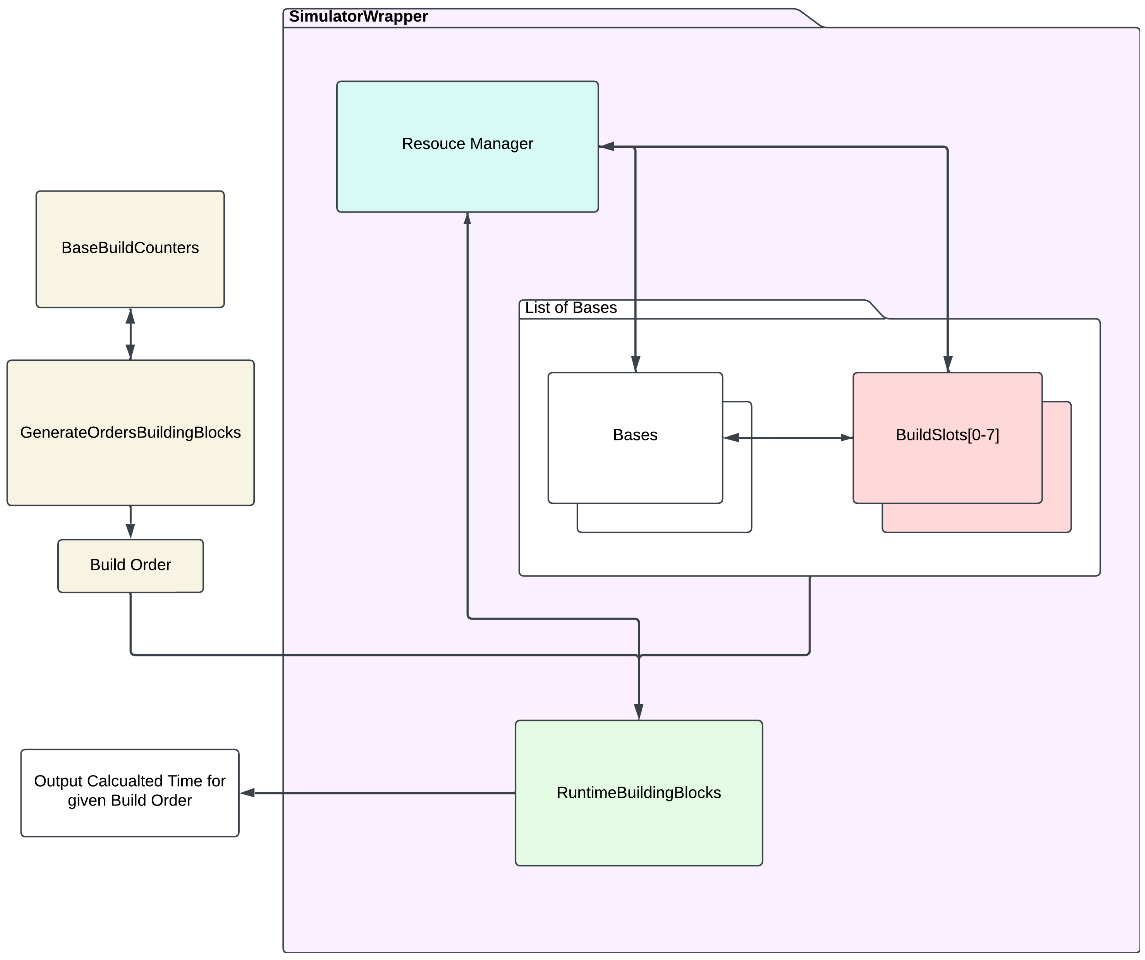


Figure 15: High-Level Overview of Class Relationships

The integration process followed these steps:

- **Build Order Generation:** The `GenerateOrdersBuildingBlocks` class would create a randomized, non-repeating build order for testing.
- **Resource Manager and Base Initialization:** The `ResourceManager` was then instantiated and linked to all `Base` and `BuildSlot` objects. These components were further referenced by the `RuntimeBuildingBlocks` class, enabling it to manage build orders effectively.
- **Simulation Wrapper:** The `SimulationWrapper` class consolidated all simulation-related objects, simplifying their initialization and providing a single interface for executing simulations.
- **Main Program Execution:** The final integration was handled by the `run_build_combinations.py` program, which served as the primary driver for the system. Its workflow included:

    - Initializing the `SimulationWrapper` and `GenerateOrdersBuildingBlocks` classes.
    - Managing constant values, such as maximum simulation time, starting resources, resource value triggers, and the number of bases, through easily adjustable parameters at the top of the file.
    - Iterating through the main simulation loop:
        * A new build order gets generated. If it is unique, the simulator calculates the time required to reach a predefined resource threshold. If not, the build order is skipped.
        * The calculated time and corresponding build order are added to a results list.
        * Repeat for as many times as defined by the `NUMBER_OF_SIMULATION_LOOPS` constant.
        * Once the simulation loop completes, results export all results to a CSV file.

- **Fail-Safe Mechanisms:** The `run_simulation` method included a maximum simulation time threshold. If the simulation exceeded this threshold, the iteration was terminated, and the system proceeded to the next build order.

This complete integration formed the foundation for an automated system capable of generating, validating, and simulating build orders at scale. Results were saved to a CSV file for further analysis, providing both calculated build times and the corresponding build orders.

In addition to the primary simulation program, a secondary utility program, `halo_wars_supply_pad_simulator.py`, was created, which allows users to input manually created build lists, such as the one shown in Figure 11. This utility was primarily used for testing and debugging purposes.

Users could adjust the `NUMBER_OF_SIMULATION_LOOPS` constant to control the number of iterations and rerun the program as needed. Future updates may include support for command-line arguments, enabling batch testing with varying parameters.

7 Results

7.1 Overview

Testing focused on three adjustable parameters: the resource threshold, the number of bases, and the number of build order iterations. The experimental design and parameters are detailed below.

7.1.1 Resource Threshold

- **3000 Resources**: The threshold to acquiring a Scarab with the assumption that supplies on map cover the two infantry units.
- **3300 Resources**: Similar to 3000-resource threshold but without the assumption the infantry cost is covered.
- **6800 Resources**: The threshold to construct two Scarabs. An additional 800 resources beyond the 6000 required for two scarabs is included to purchase the "Reinforcements" upgrade.

7.1.2 Base Limits

The number of bases ranged from one to two, as typically only one additional base is accessible to the player early in gameplay. Some testing was done with three bases, but it typically produced results only using two bases, ignoring the third.

7.1.3 Build Order Iterations

The number of iterations was the greatest unknown, it was unsure how many iterations were needed, so this value was adjusted to balance execution time and result reliability. Testing started out small at 100 iterations and incremented by factors of 10, reaching 100,000. These iteration runs would take between 5-10 minutes to execute, so the iterations were increased again to 200,000. One 500,000 iteration run was attempted, however, due to its execution time of about 45 minutes, higher iterations counts were deemed impractical for subsequent tests.

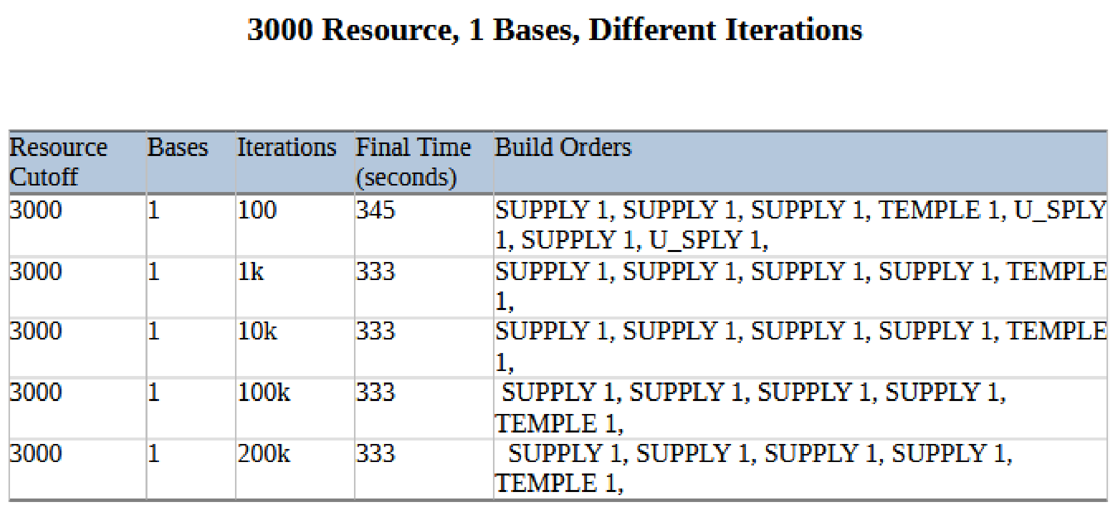
7.1.4 Build Order Terminology

The following keywords are used to describe build orders in the results tables:

- **U_BASE**: Upgrade the base, if there is no base in the base slot, build the base.
- **SUPPLY**: Build a Supply Pad.
- **U_SPLY**: Upgrade an existing Supply Pad.
- **TEMPLE**: Build a Temple.
- **1 or 2**: Numbers following keywords indicate the base to which the action corresponds.

7.2 3000 Resources Threshold

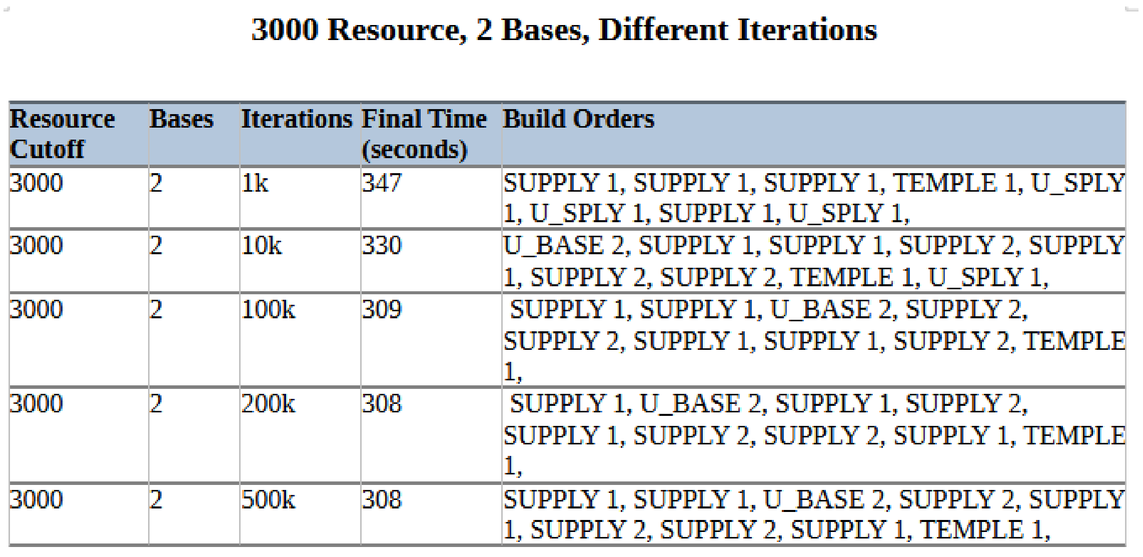
The following figures shows the results for the values gathered from the runs with one and two bases, set to a 3000 resource threshold, and variable iterations.



3000 Resource, 1 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
3000	1	100	345	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, SUPPLY 1, U_SPLY 1,
3000	1	1k	333	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3000	1	10k	333	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3000	1	100k	333	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3000	1	200k	333	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,

Figure 16: One Base 3000 Resource Threshold



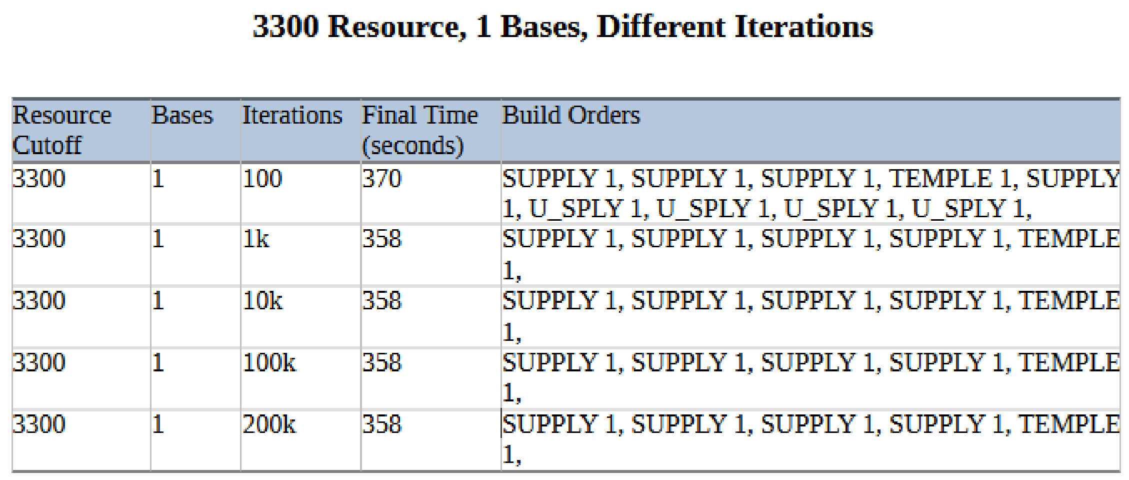
3000 Resource, 2 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
3000	2	1k	347	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, U_SPLY 1, SUPPLY 1, U_SPLY 1,
3000	2	10k	330	U_BASE 2, SUPPLY 1, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 2, TEMPLE 1, U_SPLY 1,
3000	2	100k	309	SUPPLY 1, SUPPLY 1, U_BASE 2, SUPPLY 2, SUPPLY 2, SUPPLY 1, SUPPLY 1, SUPPLY 2, TEMPLE 1,
3000	2	200k	308	SUPPLY 1, U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 2, SUPPLY 1, TEMPLE 1,
3000	2	500k	308	SUPPLY 1, SUPPLY 1, U_BASE 2, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 2, SUPPLY 1, TEMPLE 1,

Figure 17: Two Base 3000 Resource Threshold

7.3 3300 Resources Threshold

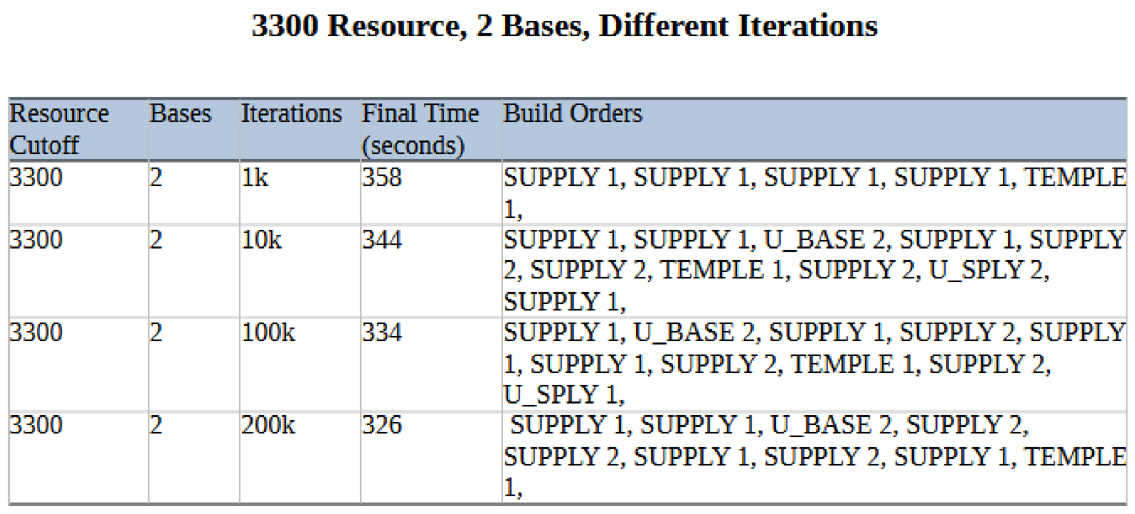
The following figures shows the results for the values gathered from the runs with one and two bases, set to a 3300 resource threshold, and variable iterations.



3300 Resource, 1 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
3300	1	100	370	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, SUPPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1,
3300	1	1k	358	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3300	1	10k	358	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3300	1	100k	358	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3300	1	200k	358	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,

Figure 18: One Base 3300 Resource Threshold



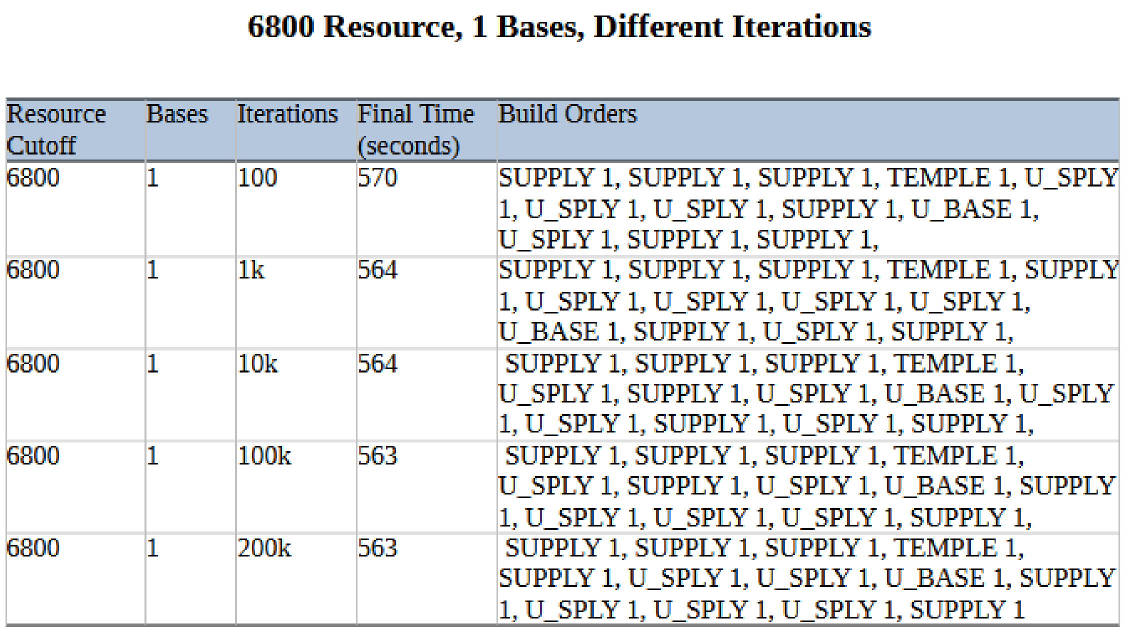
3300 Resource, 2 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
3300	2	1k	358	SUPPLY 1, SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1,
3300	2	10k	344	SUPPLY 1, SUPPLY 1, U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 2, TEMPLE 1, SUPPLY 2, U_SPLY 2, SUPPLY 1,
3300	2	100k	334	SUPPLY 1, U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 1, SUPPLY 2, TEMPLE 1, SUPPLY 2, U_SPLY 1,
3300	2	200k	326	SUPPLY 1, SUPPLY 1, U_BASE 2, SUPPLY 2, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, TEMPLE 1,

Figure 19: Two Base 3300 Resource Threshold

7.4 6800 Resources Threshold

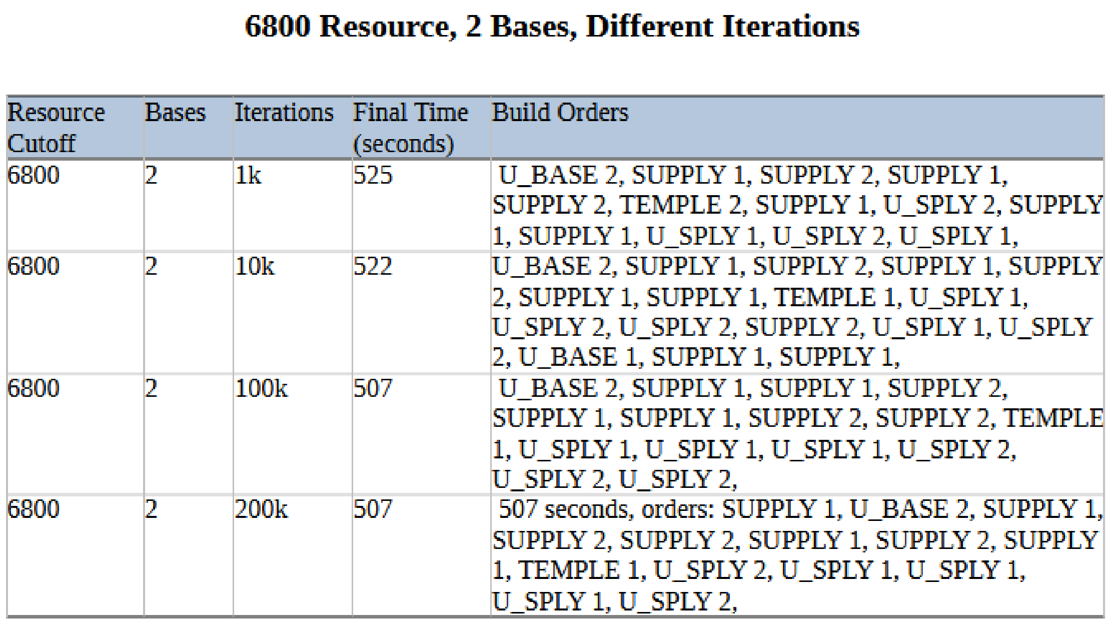
The following figures shows the results for the values gathered from the runs with one and two bases, set to a 6800 resource threshold, and variable iterations.



6800 Resource, 1 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
6800	1	100	570	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, SUPPLY 1, U_BASE 1, U_SPLY 1, SUPPLY 1, SUPPLY 1,
6800	1	1k	564	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, SUPPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, U_BASE 1, SUPPLY 1, U_SPLY 1, SUPPLY 1,
6800	1	10k	564	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, SUPPLY 1, U_SPLY 1, U_BASE 1, U_SPLY 1, U_SPLY 1, SUPPLY 1, U_SPLY 1, SUPPLY 1,
6800	1	100k	563	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, SUPPLY 1, U_SPLY 1, U_BASE 1, SUPPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, SUPPLY 1,
6800	1	200k	563	SUPPLY 1, SUPPLY 1, SUPPLY 1, TEMPLE 1, SUPPLY 1, U_SPLY 1, U_SPLY 1, U_BASE 1, SUPPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, SUPPLY 1

Figure 20: One Base 6800 Resource Threshold



6800 Resource, 2 Bases, Different Iterations

Resource Cutoff	Bases	Iterations	Final Time (seconds)	Build Orders
6800	2	1k	525	U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 2, TEMPLE 2, SUPPLY 1, U_SPLY 2, SUPPLY 1, SUPPLY 1, U_SPLY 1, U_SPLY 2, U_SPLY 1,
6800	2	10k	522	U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 1, TEMPLE 1, U_SPLY 1, U_SPLY 2, U_SPLY 2, SUPPLY 2, U_SPLY 1, U_SPLY 2, U_BASE 1, SUPPLY 1, SUPPLY 1,
6800	2	100k	507	U_BASE 2, SUPPLY 1, SUPPLY 1, SUPPLY 2, SUPPLY 1, SUPPLY 1, SUPPLY 2, SUPPLY 2, TEMPLE 1, U_SPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 2, U_SPLY 2, U_SPLY 2,
6800	2	200k	507	507 seconds, orders: SUPPLY 1, U_BASE 2, SUPPLY 1, SUPPLY 2, SUPPLY 2, SUPPLY 1, SUPPLY 2, SUPPLY 1, TEMPLE 1, U_SPLY 2, U_SPLY 1, U_SPLY 1, U_SPLY 1, U_SPLY 2,

Figure 21: Two Base 6800 Resource Threshold

7.5 Final Results

The following sections present the final results of the simulations and in-game testing for constructing one and two Scarabs, highlighting both build orders and resource timings.

7.5.1 Simulated Final Results: One Scarab

The fastest simulated result for reaching 3000 resources and constructing a Scarab with two bases was achieved in **308 seconds**. The corresponding build order was as follows:

- Build two Supply Pads on Base 1.
- Immediately construct a second base.
- Sequentially build additional Supply Pads:
  - Supply Pad on Base 2.
  - Supply Pad on Base 1.
  - Supply Pad on Base 2 (x2).
  - Supply Pad on Base 1.
- Finally, construct a Temple on Base 1.

Multiple build orders achieved the same result of 308 seconds, suggesting flexibility in the supply pad order. Key factors for success include:

- Prioritizing the construction of a second base as quickly as possible.
- Building at least seven un-upgraded Supply Pads across both bases.
- Constructing the Temple as the final step.

7.5.2 Simulated Final Results: Two Scarabs

For constructing two Scarabs, the fastest simulated result was achieved in **507 seconds** using two bases. The optimal build order was as follows:

- Build a Supply Pad on Base 1.
- Construct Base 2.
- Sequentially build additional Supply Pads:
  - Supply Pad on Base 1.
  - Supply Pads on Base 2 (x2).
  - Supply Pad on Base 1.
  - Supply Pad on Base 2.
  - Supply Pad on Base 1.
- Construct a Temple on Base 1.
- Upgrade five Supply Pads across both bases.

Similarly, exact build steps seemed to be flexible as long as the final base setup was as follows:

- Two bases with no base upgrades.
- All build slots filled with Supply Pads, plus one Temple.
- Five of these Supply Pads upgraded to heavy Supply Pads.

7.5.3 In-Game Tested Results: One Scarab

Testing the simulated build order in-game resulted in the Scarab starting construction in **306 seconds** (5 minutes, 6 seconds), which is two seconds faster than the predicted simulation result.

When additional resources were collected on the map using the scout and leader unit, the Scarab start time was reduced further to **279 seconds** (4 minutes, 39 seconds).

7.5.4 In-Game Tested Results: Two Scarabs

In-game testing for starting construction on two Scarabs revealed that while the simulated optimal build order was effective, a slightly slower alternative strategy provided better long-term resource stability. This alternative build order upgraded the second base once and constructed additional Supply Pads, leaving the player in a stronger position for late-game scenarios. This adjusted strategy had a predicted time of **509 seconds**.

Using this adjusted strategy with no extra external resources, construction started on two Scarabs in **507 seconds**, only deviating from the predicted by two seconds.

8 Discussion

8.1 Results Analysis

8.1.1 Performance Comparison: Single Base vs. Two Bases

The results from the simulation demonstrated clear patterns in the effectiveness of different strategies. One of the most significant findings was the performance disparity between single-base and two-base setups. For a single base, the fastest time to reach the 3000-resource threshold was 333 seconds, while using two bases reduced the time to 308 seconds. This represented a 25-second (7.5%) improvement, which is critical in a competitive gameplay environment where seconds can determine the outcome of a match.

8.1.2 Achieving the Stretch Goal: Two Scarabs

The stretch goal of finding the fastest build order to two scarabs was also achieved. The most optimal build order was able to produce 6800 resources in 507 seconds, which was only 67 seconds off the previous personal best for a single scarab. These results also illustrate how optimal the two base method can be, especially later in the game, as the fastest single base method took 563 seconds, almost a full 56 seconds slower.

Both of these improvements stem from the increased resource production enabled by the additional build slots available with a second base. By constructing Supply Pads across both bases and optimizing upgrades, the two-base strategy achieves a higher resource collection rate earlier in the simulation at the expense of generating fewer resources very early in the game. However, the two-base approach introduces added complexity due to the coordination required to simultaneously manage upgrades and constructions across multiple locations.

8.1.3 Algorithm Convergence and Iterations

The simulation also revealed insights into the algorithm's convergence. For single-base scenarios, the fastest times were discovered within a few thousand iterations, with little improvement observed after 100,000 iterations. This rapid convergence is likely due to the limited variability in build orders when constrained to a single base.

In contrast, the two-base strategy required significantly more iterations to approach optimal results, with meaningful improvements continuing up to 200,000 iterations. This is attributed to the larger search space created by the additional build slots and the inter-dependencies between upgrades on multiple bases.

8.1.4 Build Order Diversity

One interesting finding not directly related to the economy performance, approximately 65% of the build orders generated, for any iteration variant, were unique, leaving 35% as duplicates. While this indicates a relatively high level of diversity in the generated orders, the duplication rate also highlights an opportunity for further optimization. Reducing duplication would allow the simulation to explore a broader range of strategies within the same number of iterations, potentially leading to faster convergence on the best build orders.

8.1.5 Surpassing the Personal Best

The ultimate goal of the project was to surpass the previous personal best time of 430 seconds for building a Scarab. Both single-base and two-base strategies far exceeded this benchmark, with the two-base setup achieving a time of 308 seconds—an improvement of over 28%. This substantial reduction not only validated the effectiveness of the simulator but also underscores the potential for computational methods to uncover strategies that outperform manual experimentation.

8.1.6 Implications for Competitive Gameplay

The results demonstrate the simulator's capability to accurately predict optimal strategies under ideal conditions. In a competitive setting, a 28% reduction in build time provides a significant tactical advantage, allowing players to field a Scarab well before opponents typically expect it.

The simulator also provides a valuable framework for exploring more complex strategies, including multi-unit production and team-based optimizations, as discussed in subsequent sections.

8.1.7 Data Availability

All the CSV files and their data can be seen in the results folder in the GitHub project for further analysis.

8.2 Testing the Simulated Results vs Actual

To validate the accuracy of the simulation, several tests were conducted using the in-game engine. These tests compared the simulator's predicted build times with actual results under ideal conditions. Across multiple scenarios—including the 3000-resource, 3300-resource, and 6800-resource thresholds—the simulation demonstrated a high degree of accuracy, with discrepancies of less than 1-5 seconds in most cases.

8.2.1 Case Study: Two-Base, 3000-Resource Threshold

For example, in one test for the 3000-resource threshold using a two-base setup, the simulator predicted a build time of 308 seconds, while the actual in-game time was recorded at 306 seconds. This close alignment between simulated and real-world results highlights the simulator's reliability at modeling the Halo

Wars economy mechanics.

Additionally, the slight change could be accounted for in the fact that some supplies were gathered to cover the infantry cost, but a few resources may of been left over that resulted in a faster time to 3000 resources.

8.2.2 Case Study: Two-Base, 6800-Resource Threshold

The results for the 6800-resource threshold were also interesting, as the fastest method did indeed produce 6800 resources in the shortest amount of time, but it was not the most optimal strategy for in game performance.

Specifically, the fastest method could start construction of two Scarabs in 507 seconds, but a different method that took 3 seconds longer at 510 seconds, left the player with an upgraded second base and two more supply pads. This illustrates how as the game progresses, exact build order begins to not have as large of an impact, and simply having more supply pads earlier on will only add a few more seconds to potential builds, but result in a much healthier economy in the late game.

8.2.3 Excluded Variables for Simplicity

The testing also revealed opportunities for further optimization that were intentionally excluded from the simulation for simplicity:

- **Map Supplies**: In-game, players can collect additional resources scattered across the map. For example, clearing the reactors on the "Exile" map often provides enough supplies to offset the cost of infantry units, effectively reducing the resource threshold by 300. Collecting these supplies and accounting for their additional resources was excluded in the simulation to ensure consistency and replicability of results but offers potential for even faster builds in real-world gameplay.
- **Teammate Resource Transfers**: In multiplayer settings, teammates can send resources, significantly accelerating resource accumulation. While this was not modeled in the simulation, it presents an additional layer of strategy that could be explored in future iterations.

This method was originally tested in local games playing against an easy computer AI to ensure the strategies worked as anticipated.

8.2.4 Multiplayer and Online Gameplay Results

The build strategies were also tested in online multiplayer games. In 3v3 matches, the Scarab was consistently built before opponents could mount an effective counterattack when the enemy did not rush (a strategy where two to three leader units are sent to the opponent in the first two minutes). Without the rush, there was a 9/10 success rate in deploying the Scarab before the enemy amassed a significant army. However, the overall match win rate using this strategy only had a 3/10 success rate, highlighting the Scarab's limitations as a sole offensive option. Factors contributing to this included:

- **Counter-Strategies**: Opponents effectively countered the Scarab using air units, or by surviving late enough into the game that they could develop more powerful ground units. While the Scarabs did well in the early game, they would not be able to eliminate the enemy fast enough before opposing players could mount a successful their counter attack.
- **Resource Allocation Trade-Offs**: Focusing on building a Scarab left fewer resources for other units or defenses, making bases vulnerable to early attacks (rushes) and difficult to pivot to other units in the late game.

8.2.5 Coordinated Team Strategies

A few tests were devised where the whole team of three would all utilize the optimal build strategy for a Scarab. This was accomplished by coordinating when the teammates would swap in and out of the reactors at specific times. This produced some exciting games, and while the team experienced better success in early-game dominance, ultimately there were mixed results in the late game.

8.2.6 Summary of Simulator Performance

Overall, the simulator's performance in predicting build times was validated through rigorous real-world testing. The minor discrepancies observed between simulated and actual times could be attributed to factors such as inconsistent resource gathering from the supplies that were gathered, or by considering frame-by-frame resource updates in the game engine (60 FPS) versus the second-by-second updates in the simulator. Despite these minor differences, the simulator consistently provided actionable insights, making it a reliable tool for strategy optimization.

8.3 Further Updates and Improvements

While the project successfully achieved its objectives, several areas for improvement and expansion have been identified. Addressing these would further enhance the simulator's accuracy, efficiency, and applicability to broader scenarios.

8.3.1 Reducing Duplication in Build Orders

One notable limitation was the 35% duplication rate in generated build orders. This duplication reduces the efficiency of the simulator by consuming computational resources on redundant tests. Future improvements could include:

- Implementing smarter randomization algorithms, such as weighted probabilities based on past successful build orders.
- Introducing heuristic constraints to avoid generating build orders that are unlikely to yield new results.
- Applying techniques like genetic algorithms to iteratively refine build orders based on performance, ensuring diversity and optimization.

8.3.2 Enhancing Map-Specific Features

Currently, the simulator assumes ideal gameplay conditions on the "Exile" map, leveraging its unique advantages like uncontested base slots and free tech levels. Future iterations could expand the simulator to include:

- Map-specific constraints, such as the time and resources required to clear hostile units from additional base slots.
- The ability to simulate different maps with varying layouts, resource distributions, and reactor placements.
- Already there are enhancements in the simulator that put a "pause timer" on building the second base. This is to simulate the time needed to clear a second base. Additionally there is a flag that when enabled, will not start the pause timer until a Temple has been built, to simulate waiting for the leader unit to be generated. These are areas that could be expanded on in future updates.

8.3.3 Expanding Multi-Unit Optimization

While this project focused on optimizing build orders for the Scarab, the simulator could be extended to optimize for:

- Simultaneous production of multiple unit types (e.g., Scarabs and infantry).
- Evaluating trade-offs between building a Super Unit and assembling a more diverse army, or not building a scarab at all.
- Testing build orders for the UNSC faction, which has different mechanics and units compared to the Covenant.

8.3.4 Improving Resource Collection Mechanics

The simulator currently updates resource generation on a second-by-second basis, whereas the in-game economy updates every frame (60 frames per second). Future improvements could include:

- Adopting finer-grained updates to more closely match in-game mechanics.
- Adding variability to account for map-specific resource drops or interruptions to production.

8.3.5 Integrating Real-World Testing Feedback

Feedback from online matches indicated several opportunities to refine the simulator's strategic applications:

- Incorporating a teammate mode where simulated teammates can send resources or clear reactors.
- Adding options to account for in-game counter-strategies, such as early enemy attacks or specific unit counters to the Scarab.

8.3.6 Bug Fixes

While not major, it appears there were a few bugs that are still present in the code, mainly:

- Build orders that made it into the CSV file that only built a Temple and nothing else.
- Build orders that did not produce enough resources before the simulation time cut off was reached.
- Further investigation is required to understand how these invalid build orders were generated and to implement safeguards to prevent them.

8.3.7 Long-Term Vision

In the long term, the simulator could evolve into a general-purpose tool for optimizing strategies across other problems. Its modular design allows for:

- Customizing mechanics for different games or factions.
- Adapting to non-gaming domains, such as logistics optimization or resource management in real-world applications.

By addressing these areas, the simulator can grow from a specialized tool for Halo Wars optimization into a more robust and versatile framework capable of tackling a wider range of strategic problems.

9 Conclusion

This project was an overwhelming success, accomplishing all the goals and stretch goals set at the beginning. By reducing the fastest Scarab build time from 430 seconds to 308 seconds—a 28% improvement—this work has not only exceeded the previous personal best but also demonstrated the potential of computational methods to revolutionize strategy in Halo Wars. The simulation's accuracy, validated through real-world gameplay, proved to be within seconds of actual in-game results, showcasing the system's reliability and effectiveness.

Beyond the immediate results, this project highlights broader implications. The modular, scalable simulator framework can serve as a foundation for analyzing build orders in other real-time strategy games, enabling players and researchers to optimize resource management and decision-making. The techniques developed here, such as random order generation and build order verification, could be applied to challenges in logistics, scheduling, or even real-world resource allocation problems.

The project also served as a rich learning experience, providing opportunities to explore advanced Python programming techniques, including hash maps, singleton patterns, and combinatorics. The creation of a robust and maintainable codebase ensures that future extensions, such as incorporating UNSC factions, simulating additional maps, or testing more complex scenarios, can be implemented with relative ease.

Future improvements could include reducing duplicate build order generation, refining map-specific logic, and expanding the simulation to encompass additional game mechanics or factions. These enhancements would further increase the system's versatility and precision.

Ultimately, this project demonstrated the power of computational approaches in tackling complex strategic problems. By bridging the gap between theoretical optimization and practical validation, it opens the door to new possibilities in gaming, AI, and beyond. The journey to uncovering the fastest build order has been both rewarding and educational, offering valuable insights into programming, strategy, and problem-solving.

10 References

[1] Halo Wars Economy Discussion Board https://denkirson.proboards.com/thread/5118/halo-wars-unit-stats.

[2] Ensemble Studios. *Halo Wars*. Microsoft Game Studios, 2009. Xbox 360. Definitive Edition developed by 343 Industries and Creative Assembly, 2017. Xbox One and PC. https://www.halowaypoint.com.