Universiteit Antwerpen

An Automated Approach to Behavior Space Exploration
A Proof-of-Concept for Game Development

Joshua Moelans

Promotor: Prof. Hans Vangheluwe

Co-Promotor: Prof. Clark Verbrugge

Assistant Adviser: Randy Paredis

Dissertation Submitted in June 2024 to the
Department of Computer Science
of the Faculty of Sciences, University of Antwerp,
in Partial Fulfillment of the Requirements
for the Degree of Master of Science.

Ansymo
Antwerp Systems and Software Modelling

Contents

List of Figures

List of Tables

Acknowledgements

I want to start by thanking my promotor and co-promotor, Professors Hans Vangheluwe and Clark Verbrugge. They introduced me to the world of theoretical studies on video games, which perfectly intertwines my two interests of software engineering and gaming. Getting the freedom to explore this field for my thesis has been a real joy, which would not have happened without them.

Next, my thesis supervisor Randy deserves a thank you; without his continuous support and feedback, this thesis would have surely been way less academic and well-structured.

A lot of love and appreciation goes out to my friends, both the old-schoolers as well as the new ones who joined me during these fascinating years at university. Without you all, staying sane and motivated would have been quite the challenge.

Lastly, I want to thank my mom, dad, moeke and oma for providing financial, emotional, nutritional and unconditional support throughout my entire academic career. Thanks for letting me find my way at my own pace, and for giving me the opportunity to pursue a degree.

Abstract

In modern technology, optimizations show up everywhere. From designing systems to training artificial intelligence, some aspects will always need a search for optimal values. However, within the enormous video game industry, practical research for automated optimisation methods is rare. Often only the theory is explored, or an academic setting is provided, even though commercial games are built using commercial tools. Hence, this thesis explores both these aspects; the former is done by providing a generic optimization framework that can work with any game, simply by providing in- and outputs of the specific environment to be optimized. The latter is achieved by building a small proof-of-concept video game in the commercially viable Godot Engine, which can then be optimized by the framework. Through a series of iterative experiments, it is shown how the symbiosis of game development and automated optimization can lead to insights whilst keeping the need for manual testing low. This can lower the barrier to entry for independent game development studios by reducing the need for costly external game testing.

Nederlandstalige Samenvatting

In moderne technologie komen optimalisaties overal voor. Van het ontwerpen van systemen tot het trainen van artificiële intelligentie, er zal altijd een zoektocht naar optimale waarden bestaan. Binnen de enorme videogame-industrie is praktisch onderzoek naar geautomatiseerde optimalisatiemethoden zeldzaam. Vaak wordt alleen de theorie onderzocht of worden praktische toepassing louter in een academische setting getoond, terwijl commerciële games worden gebouwd met behulp van commerciële tools. Vandaar dat deze thesis beide aspecten onderzoekt; het eerste wordt gedaan door een generiek optimalisatie framework te ontwikkelen dat kan werken met elk spel, simpelweg door in- en outputs te voorzien van de specifieke omgeving die men wil optimaliseren. Het tweede wordt bereikt door het bouwen van een kleine proof-of-concept videogame in de commercieel toepasbare Godot Engine, die vervolgens kan worden geoptimaliseerd door het framework. Door middel van een reeks van iteratieve experimenten wordt aangetoond hoe de symbiose van spelontwikkeling en geautomatiseerde optimalisatie kan leiden tot inzichten, terwijl de nood aan handmatig testen laag blijft. Dit kan de drempel voor onafhankelijke gameontwikkelingsstudio's verlagen door de behoefte aan dure externe speltests te verminderen.

CHAPTER 1

Introduction

The domain of (embedded) systems engineering has the notion of Design Space Exploration [2], which provides an automated approach to aid designers in prototyping and optimisation of complex systems. In the field of artificial intelligence State Space Exploration exists; it has similar goals of limiting the search for an optimal solution to only relevant subsections of all possible states that can be explored [3]. This thesis proposes another method, Behaviour Space Exploration, which allows developers to analyse certain behavioural aspects of a system without having to manually test all possible cases. By providing a set of inputs as well as an output scoring function to the system, the proposed framework allows for automated exploration of the input space. Behaviour Space Exploration is not a new term, but it is relatively unexplored within the context of video games. Hackenberg and Bytschkow apply the concept as a model analysis tool, limiting model checking to heuristic and/or random results [4]. Gomes et al. discuss the approach in their paper on evolutionary robotics and how it can act as a metric for novelty search variants [5].

In this thesis, the framework is showcased by providing a Proof-of-Concept system in the form of a Video Game wherein an optimisation is to be found to solve issues and provide parameters for new features iteratively. The behaviours of the relevant parts of the system are defined in terms of statecharts [6], which help to abstract away the implementation details of the specific game engine environment that is used.

1.1 Motivation

External game testing costs money. On average between 1 and 5 percent of the total budget is spent on testing various aspects of a game during its development cycle [7]. Still, this cost is a barrier to entry for most smaller-scale productions. Since the market share of these 'indie' games keeps steadily growing (up to 31% of all Steam revenue in 2023 [8]), alternative methods to paid user testing become interesting. The back-and-forth between testers and developers on issues that might not be exclusive to the human experience reduces the necessity of human-interactive testing methods. Often these aspects are fixed by tweaking certain values or re-implementing specific parts of the game system behaviours. Hence, the idea of letting developers define these parameterised behaviours, which can be tested in an automated fashion, is proposed.

Out of this specific use case, an adaptable framework is born that acts as a general-purpose optimiser for any input-output system. To show its validity within the realm of commercial video games, a small proof-of-concept game is built alongside the framework, using a commercially viable game engine.

1.2 Contributions

In this thesis, the following main contributions are presented:

- **Optimization Framework**
  A generic and reusable input-output system that provides an interface for optimization is presented. It requires some object parameters to optimize, along with some output-processing function. For this thesis, a concrete implementation is provided; a video game optimizer built for a specific game, showcasing what kinds of in- and outputs the framework can handle.

- **Proof-of-Concept Experiments**
  Using the Optimization Framework and an example game implementation using a commercial game engine, several experiments can be performed. These provide a proof-of-concept workflow for game development which intertwines feature implementation with automated optimization. The focal point of the features to optimize is the realm of tactical communication and how this can impact character behaviour.

1.3 Technical Remarks

The proof-of-concept video game was implemented using Godot 4.2, using the GDScript[1] programming language. Inspiration for the base implementation of a top-down shooter in Godot was taken from the tutorial series by jmbiv[2]. The following image assets were used, both provided under Creative Commons CC0 by Kenney.

- **Top-down Tanks Redux**
  `https://kenney.nl/assets/top-down-tanks-redux`
- **Top-down Shooter**
  `https://kenney.nl/assets/top-down-shooter`

The optimiser framework was implemented using Python 3.10 with the following libraries:

- **NLopt**
  A free and open-source library for nonlinear optimization [9].
- **NumPy**
  A library for mathematical operations on arrays and matrices [10].
- **Matplotlib**
  A plotting library for creating static and dynamic data visualizations [11].

Performance experiments were run on a desktop PC with the Windows 11 operating system, using an AMD Ryzen 7 2700X CPU, an NVIDIA RTX 2070 Super graphics card and 32 gigabytes of DDR4 RAM.

The code for both the framework and the implemented game is provided at the following GitHub repositories:

- **Framework**
  `www.github.com/JoshuaMoelans/thesis_framework`
- **Game**
  `www.github.com/JoshuaMoelans/Master-Thesis-Godot-exploration`

[1] `https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/index.html`

[2] `https://www.youtube.com/@jmbiv_dev/`

1.4 Outline

The remainder of the thesis is structured as follows: Chapter 2 provides some related work that helped to inspire this thesis. Next, Chapter 3 outlines the process of building the actual game that is to be used during the framework experimentation phase. Chapter 4 describes the framework, both in the generic case as well as the specifics for the video game use case. Chapter 5 combines the previous two, showcasing how the iterative process of adding a feature to the game followed by automated parameter optimization is done. The final Chapter 6 summarizes the thesis and its main contributions, as well as noting some potential future work related to both the optimization and game aspects of the research.

CHAPTER 2

Related Work

The field of theoretical game studies is widely researched, yet the specific contributions made in this thesis are rarely found. A lot of analysis is done within the domain of game-solving artificial intelligence (think of AlphaGo [12] and its lineage of successors; AlphaGo Zero [13], AlphaZero [14] and MuZero [15]) as well as highly theoretical research that stays within the academic space. Some works that analyze actual game development processes exist, yet these often investigate and rank existing methods instead of presenting new ideas.

The works listed below all pertain to some aspects of this thesis. Some research is more related to potential future work, hence these entries are discussed in their relevant subsections of Chapter 6 Section 6.2.

2.1 Parameter Tuning With XVGDL+

In *"Automated video game parameter tuning with XVGDL+"* [16], Quiñones and Fernández-Leiva explore parameter optimization as a mechanism for solving artificial intelligence (AI) research problems. This is implemented into XVGDL+, an XML-inspired language for specifying video games and their desirable properties. Optimization is done through a multi-start local search algorithm using hill climbing.

The setup of their research is similar to that of this thesis. However, a big difference exists in the motivation behind the automated optimization; this thesis presents it as a tool for actual game development using a commercially viable game engine, whereas the XVGDL+ language and accompanying XGE+ engine are purely meant for research purposes.

2.2 Strategic Diversity

In the paper *"Automatic Game Tuning for Strategic Diversity"* [17] by Gaina et al. a case study is performed studying the usefulness of active learning to reduce the need for playtesting when choosing an optimal set of game parameter values. Again, instead of using a commercial game engine, the specific game they implement is done so using the General Video Game AI (GVGAI) framework (which has not seen any updates since 2021). The optimization aspect is performed by a genetic algorithm, where each gene in an individual represents a single game parameter. For these, mutation leads to changes in these values between generations, which can be evaluated and lead to selection in the individuals.

They do not have generic optimization in mind but aim their sights at finding good *strategic depth* (defined as "*the ability of a level to be played by multiple AI agents biased towards different strategies, and for multiple such agents to be equivalently effective*"). How impactful this metric is for a human player's experience was measured, but yielded no significant results. In their conclusion, the limited nature of the GVGAI framework is mentioned, as well as the lack of generalizability of the strategic depth measurement. This leaves room for this thesis to evaluate similar game tuning methods in an actual fully-fledged game engine, as well as opening up the metric to optimize for to become any scoring function instead of only one measurement.

2.3 RPSPA

A less video-game-focused prior research exists, done by Kocsis et al. In their paper *"RPSPA Enhanced Parameter Optimization in games"* [18], parameter optimization for poker and LOA (a chess/checkers-like board game) is performed. However, instead of optimizing the parameters of the games themselves, the parameters of so-called game programs are enhanced. These programs are used to play the games and have parameters for their evaluation and search functions.

Even though the goal is slightly different than in other related works, the method is similar; a modified stochastic hill-climbing algorithm is used to find optimal parameter values. This research is slightly further removed from this thesis' contents, yet it shows how even outside the realm of video games a generic optimizer could be used.

2.4 Reveal-More

Chang et al. focus their research on human interaction and exploring areas within games. In *"Reveal-More: Amplifying Human Effort in Quality Assurance Testing Using Automated Exploration"* [19], they propose the Reveal-More technique; it uses game state saving and loading to use as seeds for automated exploration. The goal is to maximize coverage by doing this, reducing the necessity for costly human quality assurance (QA) testing.

This is not directly tied to the goals of this thesis, yet some interesting ideas are shown; they provide a generic tool that is game-agnostic (one only needs to provide adequate hooks for game input as well as a save-load feature) and is built to amplify the efforts of QA testing rather than wholly replace humans in the loop.

2.5 Statechart-Based AI in Practice

The research paper [20] by Dragert et al. shows a designer-friendly approach to modelling video game non-player character AI by using the statechart formalism. It also demonstrates how their approach is suitable for developing commercial game AI by implementing the behaviour of units from the *Halo* series of video games.

This approach to modelling behaviour in a game-engine-agnostic setting is also followed in this thesis, and can be seen in Section 3.2.4 of Chapter 3. In contrast to the paper, the implementation of the AI itself is done in a commercial game engine instead of Mammoth, a research framework for massively multiplayer online games (MMOs), which Dragert et al. used.

CHAPTER 3

Building the Game

To lay out a foundation for this thesis, the Proof-of-Concept game was developed first. This Chapter outlines the process, starting with the choice of both game and engine in Section 3.1. Next, a description of the game's systems is given in Section 3.2, wherein both the general design overview as seen in Section 3.2.1 as well as the unit artificial intelligence (AI) in Section 3.2.4 are provided. Lastly, it is shown how the system is prepared to slot into the framework for automated optimisation in Section 3.3.

3.1 Choice of Game and Engine

The base concept of the game itself comes from the existing problems in coordination with in-game non-player companions/characters (NPCs). This has been described by Warpefelt and Strååt [21] as a combination of the *Lack of Awareness* and NPCs having bad *Models of Others*. The former expresses how characters are either unaware or over-aware of events (for example, not reacting realistically to gunshots or being able to track enemies through walls), whilst the latter describes how NPCs know what other entities are doing and where they are (for example, having spotted an enemy moments before, but once the line of sight is broken immediately forgetting their last position). This often breaks immersion because of the difficulties in effective communication with these companions, leading to interference when there should be cooperation. In an ideal world, these non-player characters should be smart enough to not hinder a player's (or each other's) progress, for example by perfectly following voice-based commands (much like one would use when communicating with 'real' teammates).

This leads to two distinct aspects that can possibly be optimised; (1) the reactive commandeering behaviour (r.c.), and (2) the autonomous behaviour (a.b.). Since there exists a wide range of game genres, analysing them on these two criteria gives a nice overview of those genres which could be of interest. A selection of the most prominent genres was taken from Wikipedia's *List of video game genres* [22]. Most of them have self-explanatory names, with the abbreviation for Massively Multiplayer Online (MMO) being an exception. The result of this analysis can be found in Table 3.1. Each category is ranked on the overall presence of both aspects and has some listed example games from that genre. These games, though occasionally multi-genre, have been listed as exemplars within their primary genre's row.

genre	**example games**	**r.c.**	**a.b.**
shooter	Rainbow Six Siege [23], SWAT 4 [24], Ready or Not [25]	⊗	⊗
strategy	Starcraft [26], Civilization [27], Into the Breach [28]	⊕	⊗
simulation	Cities Skylines [29], Flight Simulator [30], Stardew Valley [31]	⊗	⊕
management	Rimworld [32], Frostpunk [33], Total War [34]	⊕	⊘
fighting	Street Fighter [35], Rivals of Aether [36], Tekken [37]	○	⊗
MMO	World of Warcraft [38], Black Desert Online [39], Temtem [40]	○	⊗
role-playing	Final Fantasy XV [41], The Witcher [42], Undertale [43]	○	⊗
survival	The Forest [44], ARK: Survival Evolved [45], Valheim [46]	○	⊗
puzzle	Portal [47], LIMBO [48], The Talos Principle [49]	○	⊖
platformer	Super Mario Odyssey [50], Crash Bandicoot [51], Celeste [52]	○	⊖

Table 3.1: An overview of popular video game genres with example games, and their scores on *reactive commandeering* and/or *autonomous behaviour/AI.* Amount is measured from small to large : ○ < ⊖ < ⊘ < ⊕ < ⊗.

From the Table is clear which of the genres could be used to evaluate these criteria: simulation games such as Cities Skylines [29], strategy games such as

Starcraft [26] or shooter games such as Rainbow Six Siege [23]. Other categories often lack depth in the reactive commandeering aspect, and sometimes even the autonomous behaviour is quite simple (think of enemy behaviour in any Super Mario game). Simulation games often have intricate systems that react to player input, but these are presented in very complex scenarios with extensive scale (which is out of the scope for this thesis). In strategy games the autonomous behaviour is often quite in-depth, consisting of many layers of interactive elements. Hence, the shooter category is chosen because of the more linear nature of its combat mechanics, the adequate complexity of both the commandeering and behavioural aspects, as well as the real-life application of team tactics in actual military environments [53].

From this general genre, inspiration was gathered from the tactical combat shooter subgenre of games (the 'shooter' Table entries fall within this subgenre). All of these games offer an experience where the player can engage in hostile situations alongside non-player companions. However, most of the negative feedback on these games references the lacklustre nature of the existing companion AI systems. A collection of some reviews can be found in Appendix B.1.

Instead of overcomplicating the actual game for the thesis, a simplified version of a tactical combat shooter is implemented. The game environment is built as a two-dimensional top-down world, wherein the visuals are kept basic but functional. The choice to go two-dimensional was made because adding a third dimension would only introduce unnecessary complexity and computational overhead without being relevant to the specific solutions this thesis proposes. A screenshot of the game can be seen in Figure 3.1.

The core design of the game consists of two teams of combat units, labelled as the 'allies' and the 'enemies', who exist on a grid-like map and can interact with one another through combat encounters. A player character can be introduced to allow for user input for the reactive commandeering aspect.

Instead of also overcomplicating the thesis by building a game entirely from scratch, a game engine was chosen to provide a framework to build the game in. Given the requirements of the core design, the engine of choice is Godot 4.2 [54]. Other popular engines were disregarded for several reasons: Unreal Engine [55] is mostly used for realistic-looking 3D game environments, and lacks the tools to easily develop 2D games. Unity [56] is another engine that was a potential candidate, but also has less native 2D support. Construct 3 [57] is a 2D-focused engine which provides visual rule-based programming tools, but is difficult to integrate with. RPG Maker [58] is used exclusively to develop top-down RPG

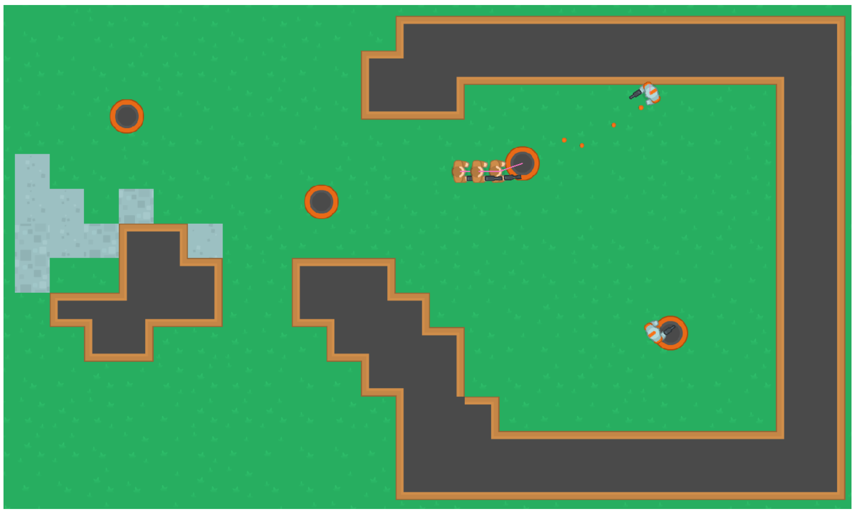
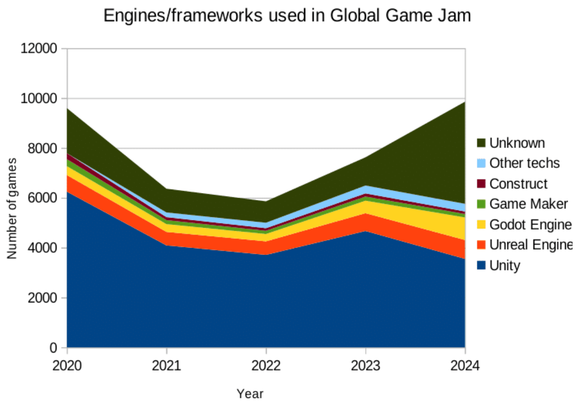
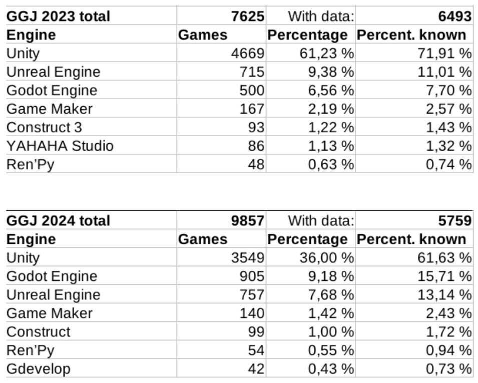


Figure 3.1: Screenshot of the game.

games, which makes it too restrictive of an engine to be used for the shooter game proof-of-concept. A strong second contender is GameMaker [59], as it provides similar functionality to Godot. However, Godot was still preferred since it combines a dedicated 2D engine with a simple yet complete toolset to create games, and has been rising in popularity [1] as can be seen in Figure 3.2. It was also chosen because of the open-source nature of the Godot Engine [60], which is helpful when debugging project-specific issues.





GGJ 2023 total	7625	With data:	6493
Engine	**Games**	**Percentage**	**Percent. known**
Unity	4669	61,23 %	71,91 %
Unreal Engine	715	9,38 %	11,01 %
Godot Engine	500	6,56 %	7,70 %
Game Maker	167	2,19 %	2,57 %
Construct 3	93	1,22 %	1,43 %
YAHAHA Studio	86	1,13 %	1,32 %
Ren'Py	48	0,63 %	0,74 %

GGJ 2024 total	9857	With data:	5759
Engine	**Games**	**Percentage**	**Percent. known**
Unity	3549	36,00 %	61,63 %
Godot Engine	905	9,18 %	15,71 %
Unreal Engine	757	7,68 %	13,14 %
Game Maker	140	1,42 %	2,43 %
Construct	99	1,00 %	1,72 %
Ren'Py	54	0,55 %	0,94 %
Gdevelop	42	0,43 %	0,73 %

Figure 3.2: Game engine popularity in the Global Game Jam. Adapted from [1].

3.1.1 Introduction to Godot

Since explaining the game's development process contains some Godot-specific terminology, this Section is dedicated to a brief introduction to game development using the engine. The information below has been gathered from the official Godot documentation [61] and contains the most relevant items necessary to build the proof-of-concept game.

Node
The generic building block that is used to make games. Combining these can be done in a **scene**. There are a lot of node types, such as Cameras, Audiolisteners, Rigidbody, Raycast, ... each with its specific purpose, methods and properties.

Scenes
A scene is a tree-like collection of these **nodes**. It can be saved to the disk and re-used, and can represent anything from a character to a weapon, an entire level or a simple user interface (UI) element.

Scene Tree
Much like **nodes** can make up a **scene**, these **scenes** can be combined into what is called a scene tree. A game consists of one **main scene tree** that can contain both **scenes** as well as **nodes**.

Script
Like in other game engines, Godot provides a way to program behaviours for the **nodes** in our **scenes**. This can be done through the use of GDScript (an imperative object-oriented programming language that looks similar to Python) or C#. These are attached to **nodes** and can be created from user-provided or built-in templates.

Signals
Nodes can emit signals whenever an event occurs. These can be used to communicate between **nodes**, for example by lowering a UI ammo counter when a player shoots their gun. They are implemented following the Observer pattern. One can define their own signals, which should follow the 'call down signal up' idea; in the scene tree, method calls should happen top-to-bottom, whilst signals should be sent bottom-to-top. This strengthens loose coupling and allows independent functioning of emitter and observer [62].

TileMap
A **node** type that is used to draw 2D tile-based maps. They use a TileSet (a tile library that can hold pixel rendering data and custom property layers such as a physics layer for collisions) to allow drawing of a game's layout.

NavigationServer2D
This provides an interface for low-level 2D navigation. It can handle navigation maps (which can be built from **TileMaps**), regions and agents. Under the hood, it uses the 3D navigation server (with the y coordinate set to 0.0) which can use navigation maps to calculate paths with an implementation of the A* algorithm[1].

3.2 Building the Game's Systems

From the general concept of the game and the choice of game engine, the actual building of the core systems can start. An overview of the design is shown in Section 3.2.1. Next, a short explanation of the recurring 'manager' structure is clarified in Section 3.2.2, followed by a description of the game's visuals in Section 3.2.3. The dynamics of the game are laid out in Section 3.2.4, followed by a brief Section on pathfinding in 3.2.5, with a finishing Section 3.2.6 on the player controls.

3.2.1 Design Overview

Figure 3.3 represents a high-level overview of the different systems in the game. One instance consists of six core elements. Most of them are 'Nodes' in Godot, except the navigation map which is a 'TileMap'.

- **NavMap** - The navigation map used to draw the tiles of the game.
- **BulletManager** - The node which handles the creation of bullets through an attached script. All spawned bullets from any weapon will become child elements of this, allowing us to track them regardless of who fired them.
- **AllyManager** - The manager responsible for keeping track of all allied units. Each of these units has a weapon and a few scripts that implement their behaviour. The manager itself has code that initialises each unit and keeps track of the number of ally deaths.

[1]`https://en.wikipedia.org/wiki/A*_search_algorithm`

- **EnemyManager** - A node which is very similar to the one above, but handling the enemy team instead. Functionality-wise, this does the exact same.
- **AdvancePoints** - This node contains a list of sequential checkpoints which the allied units use to move through the map.
- **Pathfinding** - This last node is necessary for Godot's built-in pathfinding tools. It uses the navigation map to allow the existing NavigationServer2D interface to generate paths between a given start and end point.

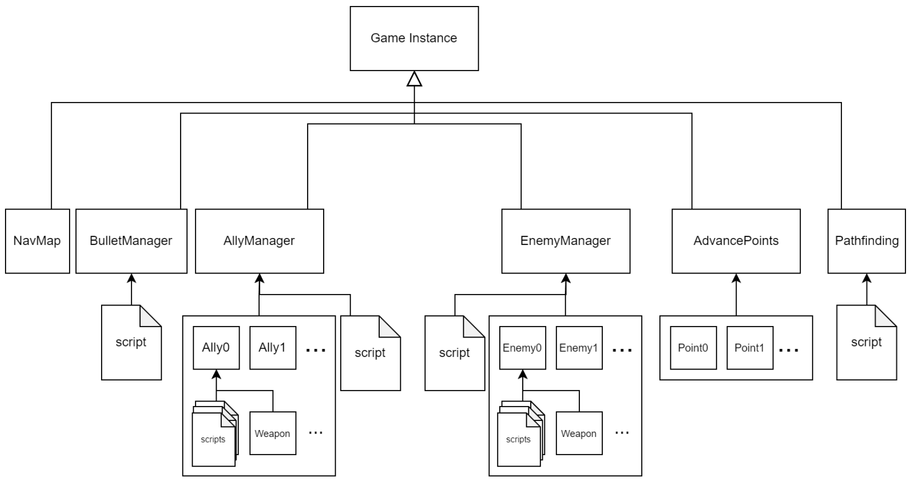


Figure 3.3: A general overview of the game design.

The dynamics of this system is described by how allies and enemies interact. The details are found in Section 3.2.4, in essence, each unit follows a behaviour chart influenced by outside events, where their goal is to move between certain positions and shoot at any encountered units from the opposing team.

3.2.2 Manager Structure

A repeating concept that is used in this design is that of the $XXXManager$. This is due to the 'call down, signal up' philosophy that dictates the way data and events should flow between scripts, as described in Section 3.1.1. In turn, this leads to lower coupling [63] and the delegation of responsibilities onto the entities that actually need the data without having to rely on in-between nodes in the scene tree.

3.2.3 Visuals

As described in the introductory Section about the approach to building the game, the visual representations of the game world are basic yet effective. A screenshot is shown in Figure 3.4 using the assets provided by Kenney.nl (see 1.3). It showcases the map with both ally and enemy units, some obstacles and a few advance checkpoints. The allied units can be seen approaching from the left side, following the circular orange advance checkpoints into the enemy enclosure. The pink lines represent the current paths they are taking towards their next checkpoint. The enemies, having a more robot-like appearance, are seen patrolling the inside of the confined square on the right.

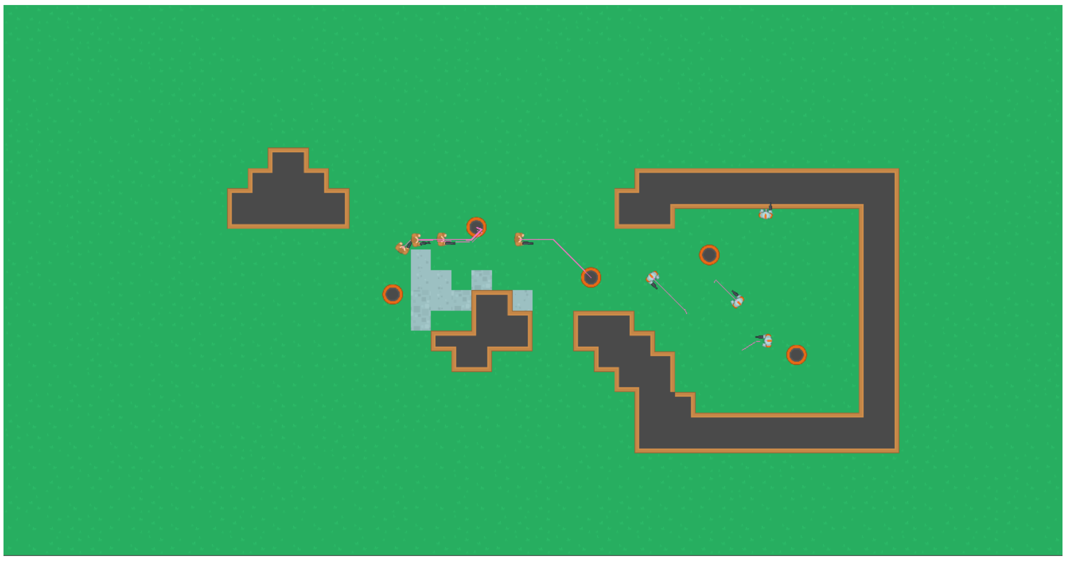


Figure 3.4: Screenshot of the game.

3.2.4 Unit AI

As described above, the game dynamics are executed by two opposing teams: the *allies* and the *enemies*. Except for the naming difference, these two behave very similarly. Hence, this behaviour can be described entirely with one statechart, as shown in Figure 3.5. It uses the formalism introduced by Harel [6] (and proven to be viable in a video game context by Dragert et al. [20]). The illustration has states (yellow rounded rectangles), composite states (white rectangles, used for the actual behaviours they represent) and transitions between them (arrows pointing from one state to the next). These transitions are labelled with the event that triggers them (empty if automatically taken) followed by a '/' and an action to perform. A diamond indicates a choice, and the circle with 'H*' indicates a deep history state, which means that a transition pointing to that circle will go back to the last state that was visited before going into the compound state it is attached to.

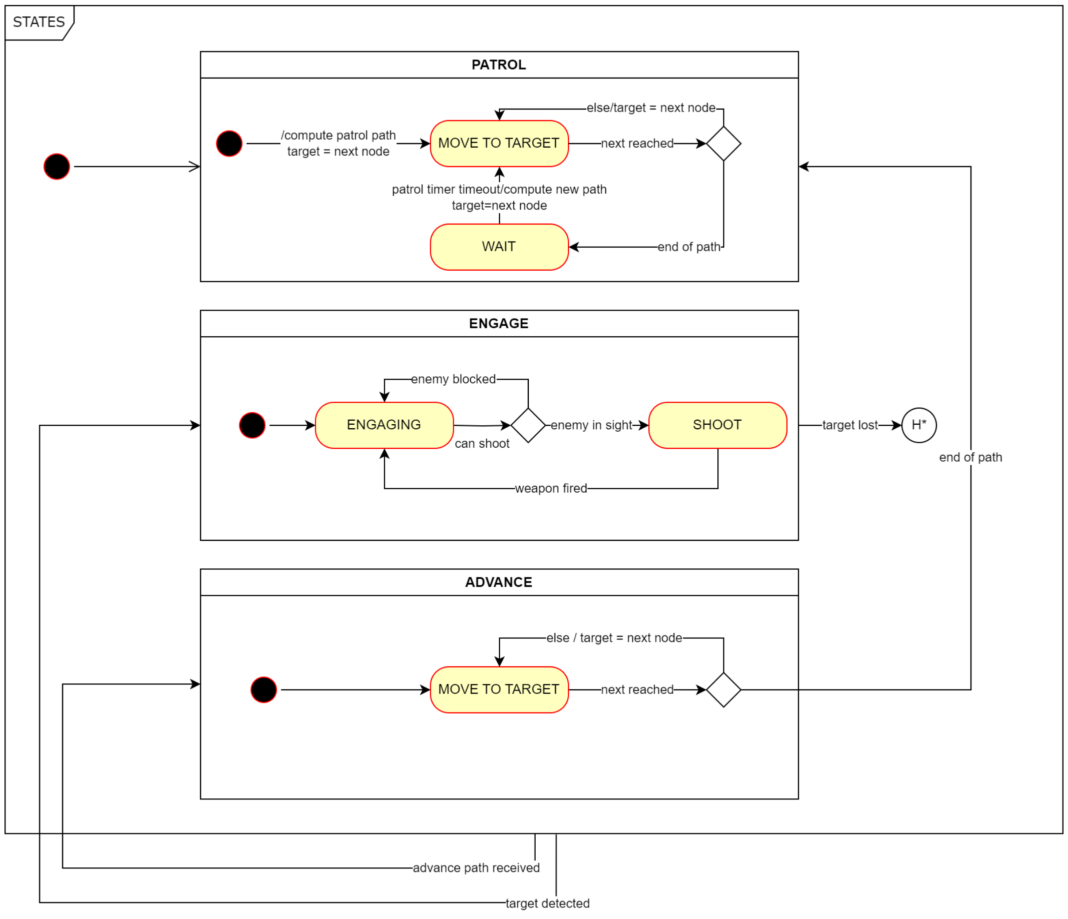


Figure 3.5: Statechart of unit AI.

The initial state for all units is the **PATROL** state. Therein a loop is executed where a patrol path is computed and the next target position is set. Once this next position is reached, a check is performed whether the end of the generated path is reached. If this is the case, a timer starts counting down (to simulate the unit scanning whether that position is safe) after which a new path is generated. If the path is not finished, the next position on that path is set as the new target position for that unit.

A similar yet slightly different state is the **ADVANCE** case. There a path is received from an outside control center, which is followed in much the same way that the **PATROL** path is to be followed. Once the path is completed, the state reverts to the base **PATROL** state again. If the given advance points list is non-empty upon game initialisation, the allied units will go to this state immediately.

Another possible external event that occurs is the *target_detected* one. Once this fires, a unit immediately goes into the **ENGAGE** state. There, as long as the target is not lost, the unit checks whether the enemy is in sight or not.

If it is, the unit shoots. If it is not, after some implicit delay, the check is repeated. Once the target is indeed lost (either by movement or by death) the engaging unit's state is restored back to the previous one.

The implementation details of the actual behaviours are left out to allow for the reusability of the same diagram for other programming languages (see Section 6.2.2). Other details, such as weapon firing and reloading, movement mechanics or the receiving of events are left out for readability purposes. These can be found in the public repository referenced in Section 1.3.

3.2.5 Pathfinding

An important part of functionality in any game involving non-player character movement is that of pathfinding. In this game, this is done following a grid-based approach, making use of the NavMap to draw which tiles are walkable areas and which ones are not. This data is then used by the built-in pathfinding interface to generate paths between any given start and end point.

During testing, it was noticed how these paths can sometimes clash, leading to units bumping into one another or even getting stuck. This is a tricky problem to solve, and has puzzled many game developers that came before [64]. As multi-agent path planning is a PSPACE-hard problem [65], in the interest of the thesis, it was decided not to spend too much time on fixing these issues. In general, a trade-off between computation and accuracy is always required. For the purposes of the game, computational performance was favoured. Hence, the paths between any start and end point are only calculated once, and not updated at every game tick. This can lead to slight inaccuracies and collisions but allows us to run the game smoothly. Some other approaches to solve this problem have been presented before. The one by David Silver in 2005 is discussed in Section 6.2.3 of the Future Work Chapter.

3.2.6 Player Controls

Alongside the autonomous unit behaviour, player controls are an important part of games. These are less interesting when running automated optimisations on the system, but can act as a sanity check for the actual playability of the game prototype. The player can do similar actions as the units, albeit less constrained by the grid-based level layout. Movement is possible in 8 directions (both cardinal directions as well as the diagonals between them) and shooting happens towards the cursor's on-screen position. Reloading can be done manually or happens by itself when trying to shoot with an empty magazine. Another feature that only the player has is that of commandeering units; by right-clicking it sends a notification to all allied units to make their way towards the cursor position. For the units, this equates to receiving the *advance_path_received* event in their behaviour statechart.

3.3 Simulation Preparation

Before being able to run the simulation phase of the optimisation loop on the created game, some additional preparatory steps are required. The human interaction aspect is removed for now, since large-scale simulation that requires user input to function would defeat the purpose of the automated optimiser framework. For validating the optimal outcome generated by the optimiser, the 'player' character will be re-placed (i.e. added back) inside of the game environment for playability testing. For simulation, it is simply replaced by a camera controller that can view the entire game scene, move around and zoom in and out.

As mentioned before, the framework requires both inputs to the system as well as output values that show how good an iteration was. The latter is done through collecting data via the game's state, which is discussed in Section 3.3.1, whilst the former becomes a part of the unit AI systems. The act of saving and loading this state is briefly introduced in Section 3.3.2. To be able to run the framework as efficiently as possible, learning the most in the least amount of time, instancing as well as headless running are used. These are detailed further in Sections 3.3.3 and 3.3.4. To compare the performance of these alterations to the game, some experiments were performed. The results thereof are shown in Section 3.3.5.

3.3.1 Game State

To be able to gather data from one single game execution a structured approach is required. Hence the concept of State is introduced, which is implemented through the storing of relevant data in a JSON-like structure. The data is gathered through the use of the aforementioned *managers* (see Section 3.2.2) that can signal the necessary data upwards. The state holds the following details for both allied and enemy units:

- *aim_direction* - Used to know where a unit is shooting towards.
- *ammo* - Representing the number of bullets left in the unit's weapon.
- *goal_position* - The target position the unit is currently trying to get to.
- *health* - A value showing the health of the unit. If this is zero, the unit is presumed dead.
- *id* - A unique identifier of that unit.
- *initial_locations* - The set of advance goals the unit follows.

- *path* - The current path from the unit's position towards the goal position.
- *position* - The current unit position.
- *previous_state* - Used to implement the Deep History from the statechart when a unit exits the engage state.
- *reload_count* - The number of reloads done by the unit.
- *state* - The unit's current state in the behaviour statechart.
- *target* - A reference to the current target's identifier.

Furthermore, all currently active bullets in the scene are tracked. For each of them, the direction of travel, position, speed and the team that the bullet originated from are stored. The amount of damage done by each team is also saved, as well as the friendly fire damage (labelled as *team_damage*). Lastly, a timer value is used as the unique identifier of the state update.

An example output can be found in the Appendix Section A.1.

3.3.2 Saving and Loading

Since the game updates these State values continuously, always storing it to a file is very resource-intensive. Luckily, constantly outputting this state is not required for large-scale simulation. However, it is interesting to re-run a specific portion of a stored game state with differing parameters to compare the results visually. A save-and-load system was implemented to allow just this, by providing an on-demand way to store the current game state or restore the game state from one of the previously saved state files.

3.3.3 Instancing

Since our game setup is quite simplistic in nature, speeding up the learning process of our optimisation framework can be done through so-called instancing. This means that one game instance is 'duplicated' in some way to allow for multiple runs of the same (or different) input parameters to parallelise the gathering of results. Since the previous Section showed how the game state can be stored (and restored) multiplication of game instances can easily be tracked by saving and loading each instance's state.

The following Sections outline two different approaches of instancing as well as a performance comparison to see what kinds of speed-up is achieved.

3.3.4 In-Game vs. Across-Game

The first type of instancing will be called 'In-Game instancing'. Here a square grid is generated of at most $n \times n$ game scenes. This grid does not necessarily need to be full; for example, a grid of 7 instances does not fit in a 2×2 square grid and does not fill up a 3×3 grid. Hence it will start 'filling up' the 3×3 grid until the last row and will keep the last two spots open. A full grid is shown in Figure 3.6 and a non-full grid is shown in Figure 3.7.

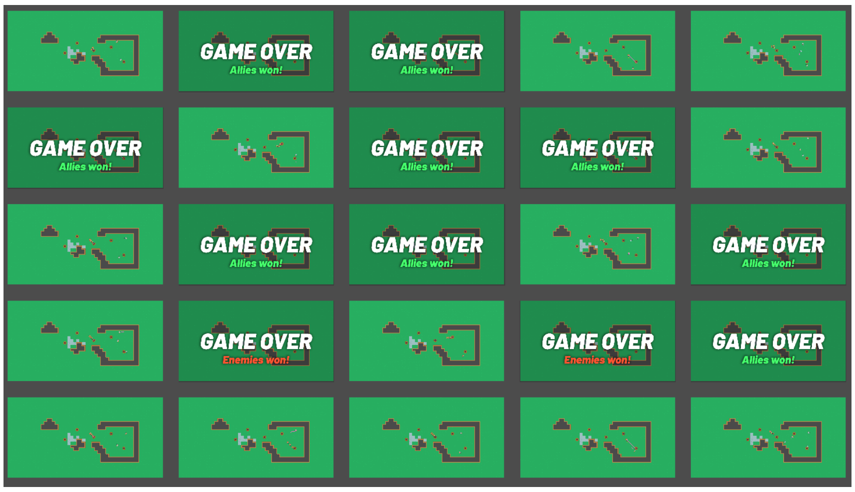


Figure 3.6: In-game instancing showing a full grid of $5 \times 5 = 25$ game scenes.

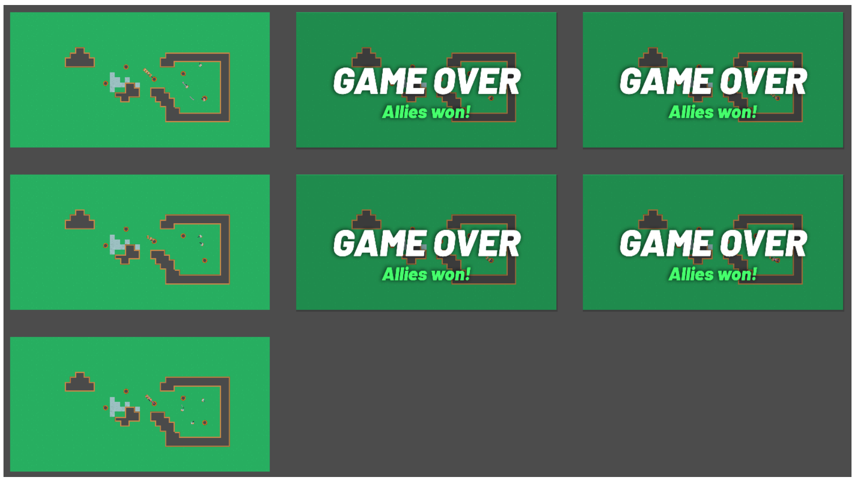


Figure 3.7: In-game instancing showing a non-full grid of 7 game scenes.

The second type of instancing is to be called 'Across-Game instancing'. This is an even simpler form of multiplying the number of game scenes, which is achieved by running multiple compiled versions of the entire game (which can then still use In-Game instancing) next to one another. An example is shown in Figure 3.8.

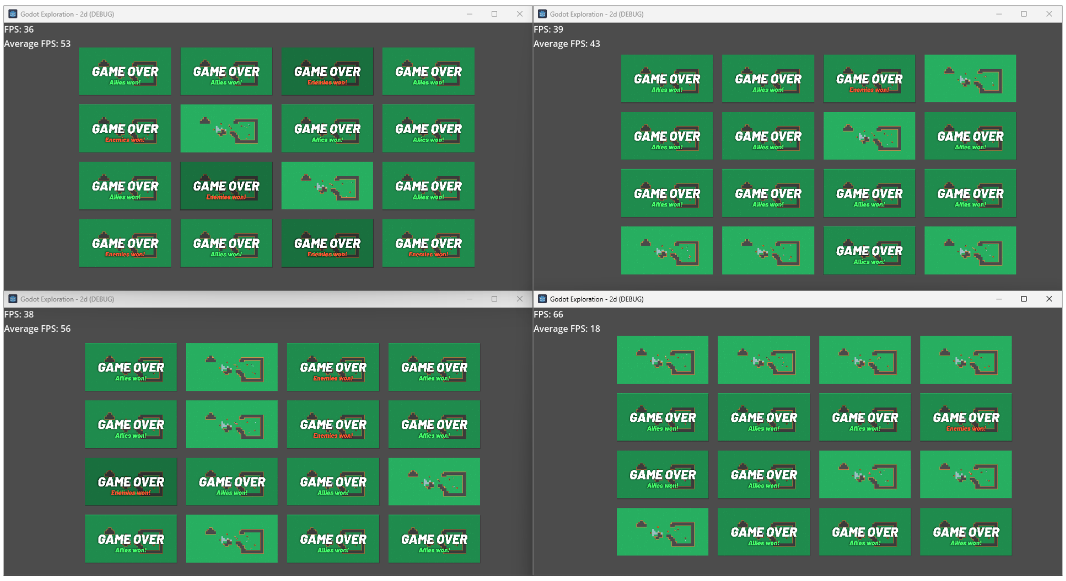

Godot Exploration - 2d (DEBUG)
FPS: 36
Average FPS: 53
Godot Exploration - 2d (DEBUG)
FPS: 39
Average FPS: 43
Godot Exploration - 2d (DEBUG)
FPS: 38
Average FPS: 56
Godot Exploration - 2d (DEBUG)
FPS: 66
Average FPS: 18
GAME OVER
Allies won!
Enemies won!

Figure 3.8: Across-game instancing showing four compiled games running side-by-side.

3.3.5 Performance Comparison

Even though the game uses simple graphics, a large chunk of computing power is used to visualise what is happening on screen. Luckily no aspects of our simulation have an intrinsic need for constant visualisations, hence an experiment was conducted to compare the average frames per second (FPS) when rendering the full grid, rendering a small subsection, or fully disabling rendering. This average was calculated by using the Godot Engine's built-in *get_frames_per_second* method. The results are shown in Table 3.2, with a corresponding visual representation in Figure 3.9.

instances	average FPS (zoomed out)	average FPS (zoomed in)	average FPS (disable rendering)
1	1400	1900	2900
2	1000	1500	2800
4	600	1300	2700
8	300	1400	2500
16	140	1000	2200
25	90	500	1900
32	35	200	1400
49	10	50*	800*

Table 3.2: In-game multi-instance performance comparison. Higher average Frames Per Second (FPS) is better. Entries with a * have higher variance in FPS, with frame hops and frame drops (going far above/below the average).

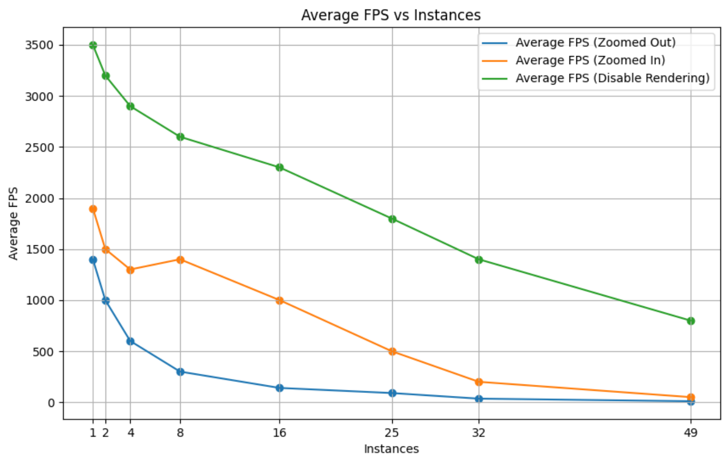


Figure 3.9: In-game multi-instance performance comparison plot. Higher average Frames Per Second (FPS) is better.

As expected, a lot less computing power is used when reducing the amount of rendered objects. Entirely disabling rendering has the biggest performance boost, with a noticeably smaller drop-off in average frames than just zooming into the scene. In terms of the median FPS, an optimum seems to be found around 25 instances, since a larger grid suffers from frame drops as well as frame hops even with rendering disabled (e.g., in the 49 instance case the FPS goes anywhere from 10 to 1500). For simulation purposes, a grid of at most 5x5 in-game instances will be used.

On top of this parallelisation technique, multiple game windows can be created to run these grid-instanced games next to one another. There is no major performance hit when running these across-game instances; a drop-off of about 50% occurs when using the built-in Debug mode to run 4 games side-by-side. Since 4 times as many games are run, yet the drop in FPS does not make each game unusable, this is another viable option to increase simulation batch size. A comparison of the FPS going from one singular game to running two or four at the same time is shown in Table 3.3. It is noteworthy that for in-game instance counts from one to eight the average FPS stays consistently at 1200, with the first minor drop appearing when increasing the in-game instance count to 16. Increasing the count to 25 drops the average down even lower, which shows that for this style of instancing an optimum sits around the 16 instance mark.

instances	**average FPS (1 game)**	**average FPS (2 games)**	**average FPS (4 games)**
1	2900	2000	1200
2	2800	1900	1200
4	2700	1800	1200
8	2500	1700	1200
16	2200	1600	1000
25	1900	1300	700

Table 3.3: Across-game multi-instance performance comparison. Higher average Frames Per Second (FPS) is better.

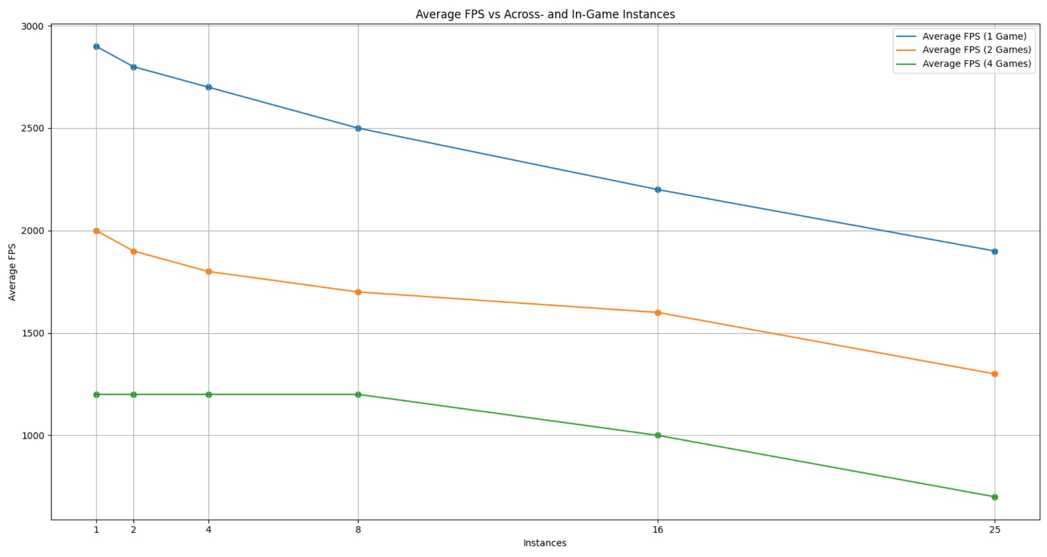


Figure 3.10: Across-game multi-instance performance comparison plot. Higher average Frames Per Second (FPS) is better.

CHAPTER 4

Building the Framework

This Chapter starts by giving an overview of the Framework approach for optimizing the game in Section 4.1. Next, the details for building a Generic Optimization Framework are given in Section 4.2. It is then shown how this framework can be used for the specific Game Improvement Proof-of-Concept in Section 4.3.

4.1 Framework Approach

During the entire development of any game, there exist two major time- and resource-consuming stages [66]. The incubation stage where a lot of creative work is done, ideas are gathered and a game vision is created, followed by the second main stage being the actual game's production. During this second phase a lot of time is spent tweaking gameplay parameters and implementing features that were designed during the incubation phase. This part of the development process often suffers from clichés such as the 90-90 rule ("... *90 percent of the game accounts for the first 90 percent of the development time. The remaining 10 percent of the game accounts for the other 90 percent of the development time.*" [67]) or the concept of 'development hell' (where development is stalled due to a number of reasons, from overambitious scope to poor development time management [68]).

The goal of the Framework Approach is to provide a tool to aid during this treacherous development phase, assisting developers by automating the parameter-tweaking and testing aspect of the production cycle. The developed framework is designed to be a broadly applicable architecture for automated optimization. This is achieved by first having it take in a set of parameters to optimize. Then, it is also attached to some outside code/environment used to gather a resulting score for an optimization iteration. Since in a video game context such parameters already exist, adapting the game to slot into the automated optimizer framework is a relatively straightforward task (i.e. by providing an interface to the game that sets parameter values when the game is loaded). Since the optimization is also based on scoring metrics, these should also be generated by the game in some way. Again, this is not an insurmountable undertaking, since games usually have a save-and-load feature that can be repurposed to provide values for scoring an optimization iteration.

4.2 Generic Framework

The reusability aspect comes from the design of the Generic Optimizer Framework. For this thesis it was implemented with the Python language, using the NLopt library (an open source toolbox for nonlinear optimization [9]). On top of the base optimization loop, a generic parameter class was designed. This provides a reusable interface for getting and setting values of the parameters that are being optimized, independent of the problem that they are being used to solve.

4.2.1 Generic Parameters

To have some values to optimize, the Generic Parameters class was designed as a reusable abstract interface for any number of parameters. As can be seen in Figure 4.1, a complementary class for any individual Generic Parameter has been introduced. Each parameter that instantiates that class is given a parameter-type, a minimum and maximum value as well as the initial value and a step size. The collective Generic Parameters class can then be implemented for any given problem by inheriting from the class and providing a list of named parameter variables of the Generic Parameter type. The abstract base class still takes care of setting and getting attributes, as well as getting the Generic Parameter member variables' values.

4.2.2 Generic Optimizer

The generic optimizer class follows the basic setup that any Python-based NLopt class should adopt. As can be seen in the class diagram on Figure 4.2, the optimizer has only three methods and three member variables. These variables are the primary optimization algorithm that NLopt should use (*opt_algo*),

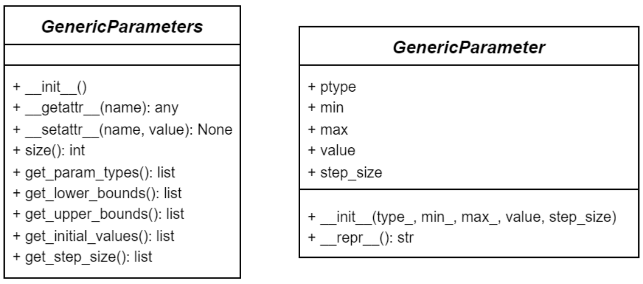

GenericParameters
+ __init__()
+ __getattr__(name): any
+ __setattr__(name, value): None
+ size(): int
+ get_param_types(): list
+ get_lower_bounds(): list
+ get_upper_bounds(): list
+ get_initial_values(): list
+ get_step_size(): list
GenericParameter
+ ptype
+ min
+ max
+ value
+ step_size
+ __init__(type_, min_, max_, value, step_size)
+ __repr__(): str

Figure 4.1: Class diagram of Generic Parameters and Generic Parameter classes.

a possible secondary algorithm (*opt_algo_2*) that is required for some primary optimization algorithms, and a dictionary (*NLopt_return_codes*) that maps NLopt return codes to more verbose messages (which can tell the user why an optimization loop has finished, for example through reaching a goal value, maximal number of iterations, running out of time, ...).

The methods consist of a constructor, which sets up the optimizer using the two given algorithms, followed by the objective function used in the optimization loop and the optimize function itself. This last method takes in three distinct stopping criteria, being the relative tolerance on parameters (xtol_rel), the relative tolerance on the objective function output (ftol_rel) and the maximum number of evaluations to perform (maxeval). These relative tolerances are used to provide an early end to the optimization loop, the first doing so when the difference in parameter values from one iteration to the next becomes too small, whilst the second does the same for the objective function. The method also uses the given parameters to set all of their lower bounds, upper bounds, step sizes and initial values. After the setup phase is done, the NLopt library does its work and performs the optimization with all of the given values. After it is finished, the output shows how long the optimization took, how many iterations were performed, and for each of the parameters what their optimal value is.

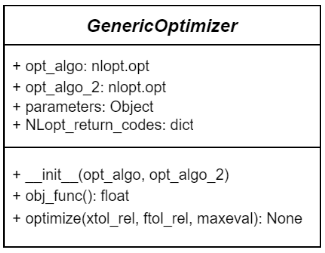

GenericOptimizer
+ opt_algo: nlopt.opt
+ opt_algo_2: nlopt.opt
+ parameters: Object
+ NLopt_return_codes: dict
+ __init__(opt_algo, opt_algo_2)
+ obj_func(): float
+ optimize(xtol_rel, ftol_rel, maxeval): None

Figure 4.2: Class diagram of the Generic Optimizer.

4.3 Game Improvement Proof-of-Concept

For the base of the game that was finished in the previous Chapter, a derived Optimizer class can be set up, inheriting from the Generic Optimizer class from the previous Section. This is done to simulate the process of automated optimization for in-game parameters as described in 4.1, which requires both simulation parameters and a way to handle simulation output. In this Section, both aspects are explained in terms of the specific top-down shooter game, by first showing an example of some optimization parameters in Section 4.3.1, followed by a description of the implementation of simulation output in Section 4.3.2. Lastly, the optimization loop including the running of the game with the given iteration's parameters is described in Section 4.3.3. The optimization algorithms and the choice thereof are shortly detailed in Section 4.3.4.

4.3.1 Simulation Parameters

As described in Section 4.2.1 any optimization needs a set of parameters. As a starting point for the game optimization, the GameParameters class was created. This class is used to represent values of in-game parameters that can be tweaked for gameplay purposes (e.g., movement speed, reaction time, bullet damage, ...). An example definition can be seen in Figure 4.3, containing two parameters that will be used in the first optimization experiment (see Section 5.1).

```python
from Parameters import GenericParameters, GenericParameter
class GameParameters(GenericParameters):
    def __init__(self):
        self.communication_count = GenericParameter(type_="int", min_=0, max_=5,
                                                    value=2, step_size=1)
        self.communication_delay = GenericParameter(type_="float", min_=0.0, max_=1.5,
                                                    value=0.2, step_size=0.1)
```

Figure 4.3: Python code showing an example GameParameters class definition.

Any number of named variables can be added to the constructor, all being instances of the generic parameter class. These get given a type, minimum and maximum values, an initial value as well as a step size.

4.3.2 Simulation Output

Like described in the previous Chapter under Section 3.3.1, the game provides the option to retrieve the current state which gets stored to a JSON file. This state can be used in the Framework to compute an objective function, by providing a weighted sum of all relevant attributes. It is up to the user to define how each game is scored. In the case of the current game setup, the first and simplest value to optimize is the amount of team damage dealt by all the allied units, which is represented in the abbreviated state output as can be seen in Figure 4.4 (for a full output of the state, see Appendix A.1). This scoring is then defined in the code as can be seen in Figure 4.5. The score method gets called to generate an output for the objective function, which NLopt uses to measure an optimization iteration's performance.

```
{
    "allies": {
        ...
    },"bullets": {
        ...
    },"damage_done": {
        ...
    },"enemies": {
        ...
    },"team_damage":{
        "allies" : 120,
        "enemies" : 40
    },"timer": 23.567812
}
```

Figure 4.4: Abbreviated state used for basic game scoring.

```python
def score_game(self, game_results:dict) -> float:
    return game_results["team_damage"]["allies"]

def score(self, results:dict) -> float:
    instance_scores =[]
    for instance_result in results.values():
        instance_score = self.score_game(instance_result)
        instance_scores.append(instance_score)
    return np.mean(np.array(instance_scores))
```

Figure 4.5: Python code for basic game scoring.

The game results are gathered by reading all of the final state files of each in-game instance. The results of each instance then get scored by passing the *instance_result* to the *score_game* method, where in this example the instance score is calculated by simply reading out the team damage done by the allies. This instance result is added to a list, such that in the end the mean value can be returned as the final score of this optimization iteration.

4.3.3 Derived Framework

To run an Optimizer on the created game, some additional steps are implemented on top of the Generic Optimizer. As can be seen in the class diagram in Figure 4.6 there are some extra variables: the location of the game is used to run the executable, the location of the logs is used to gather the simulation results, the number of in-game instances can be set (to speed up the learning of the optimizer, by running multiple games with the same parameter setup at once) as well as a timeout value (after which a game simulation should be forced to end). As described above, the parameters to optimize are given by instantiating a GameParameters object in the constructor. The last variables in the list are all used for logging per-iteration data and generating plots thereof.

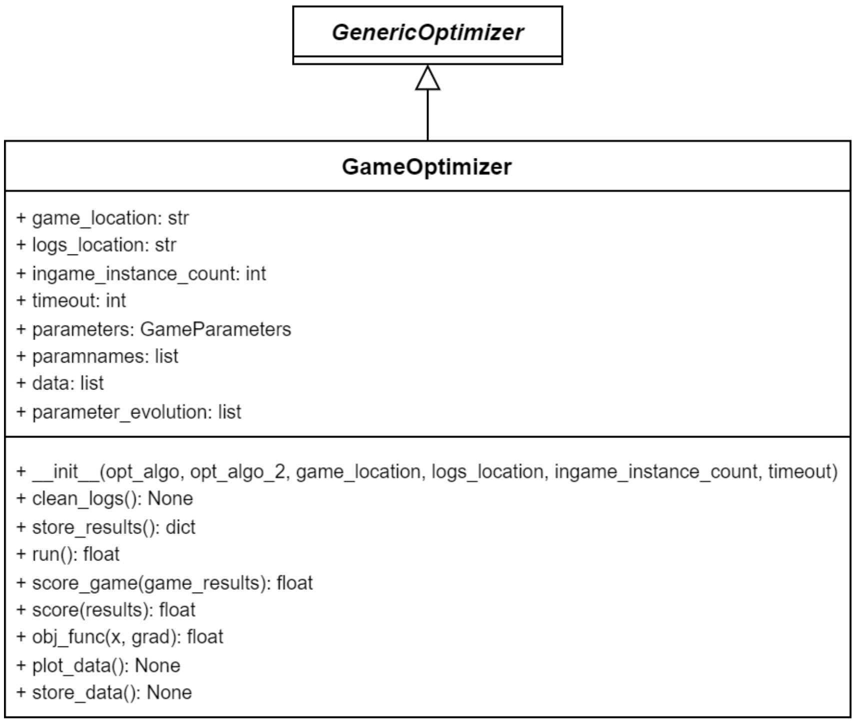

GenericOptimizer
GameOptimizer
+ game_location: str
+ logs_location: str
+ ingame_instance_count: int
+ timeout: int
+ parameters: GameParameters
+ paramnames: list
+ data: list
+ parameter_evolution: list
+ __init__(opt_algo, opt_algo_2, game_location, logs_location, ingame_instance_count, timeout)
+ clean_logs(): None
+ store_results(): dict
+ run(): float
+ score_game(game_results): float
+ score(results): float
+ obj_func(x, grad): float
+ plot_data(): None
+ store_data(): None

Figure 4.6: Class diagram of the Game Optimizer (inheriting from the Generic Optimizer).

In terms of methods, the Game Optimizer's logic is the same as the generic one; an objective function is defined, which gets called in the superclass' optimize function. This objective function calls the run method to actually execute the game with the new parameter values generated by NLopt. These are passed as command line arguments (CLA) which get parsed by the compiled game. A CLA handler exists within the game's main script, passing the arguments down to wherever they are to be used. An example code snippet that runs the game is shown in Figure 4.7, wherein the number of in-game instances is passed via *ngames*, the visibility of the games is set to true by the *visible* argument, and the communication count and delay are passed by their respective parameters.

```
subprocess.run(["game.exe","-ngames=9","-visible=true",
               "-communication_count=1","-communication_delay=0.2"])
```

Figure 4.7: Python subprocess call to run the game with given arguments.

Each in-game instance will write a final state to a file when all units of either the allies or the enemies are dead, or after the global timeout runs out. When the entirety of the game has finished, the results of all instances are collected by going through the given logs folder in alphanumerical order, and storing the *game_instance_X_GAMEOVER* or *game_instance_X_TIMEOUT* file's content for each instance X (where the *GAMEOVER* file has priority over *TIMEOUT* due to the alphabetical order). An example file hierarchy is shown in Figure 4.8.

```
logs
|--- game_instance_0_GAMEOVER.txt
|--- game_instance_0_TIMEOUT.txt
|--- game_instance_1_GAMEOVER.txt
|--- game_instance_1_TIMEOUT.txt
|--- game_instance_2_GAMEOVER.txt
|--- game_instance_2_TIMEOUT.txt
...
```

Figure 4.8: Game logs folder structure.

It is to be noted that in some cases both teams have no remaining units, since the bullets that were fired just before a unit's death do not despawn (just like in real life). In this case, the *GAMEOVER* file for that instance is overwritten by a new *GAMEOVER* file whenever the last member of the last team dies. Hence, no complex system is required to await all fired bullets to despawn before storing the final state. This data then gets used in the scoring as described above. During each run of the optimizer, the in-between data is stored and can be plotted or stored after the optimizer is finished by calling the respective *plot_data* and *store_data* methods.

4.3.4 Optimization Algorithms

A combination of global and local search is used for all game optimizations. For this, the NLopt implementations of MLSL [69] and COBYLA [70] are set as the primary and secondary optimization algorithms. The former provides a strategy of global optimization by performing a sequence of local optimizations from random starting points, where each of these local searches are done by the COBYLA algorithm. Global search is used because of the often large search space that exists by having many tweakable number-based parameters, whilst COBYLA provides a gradient-free local optimization algorithm (since the game environment does not come with a simple derivate to take advantage of, as is the case in Neural Network-like situations). COBYLA shows good performance on constrained optimization problems [71] and separates itself from other NLopt alternatives by providing different parameter step sizes [9] which are useful for optimizing differently scaled in-game parameters at the same time.

CHAPTER 5

Game Optimizer Experiments

The game as described in the Chapter 3 lacks finalized dynamics to make it feel like a truly complete experience. In this Chapter, the goal is to add more behaviours and strategies to the game, as well as fine-tune them by using the Optimizer Framework from the previous Chapter 4. The final result should feel more 'alive' than the bare-bones implementation, which is achieved by a short iteration loop that uses the power of the Framework to test new and updated features.

Each Section entails a certain optimization experiment, going from a specific problem that is to be solved, followed by a hypothesis about what changes should be made in the game to find a solution to the problem and what the optimizer is expected to find, ending with some visualized optimization results. The experiments are performed on the same hardware used in the game performance testing. Their specifications can be found in Section 1.3.

5.1 Basic Communication Strategy

In the game from Chapter 3, only basic behaviours following the statechart in Section 3.2.4 exist. One of the first additions to make the game feel more alive is a basic communication strategy. Currently, the behaviour for every unit is to just shoot in the direction of any enemy once they spot them. In Godot, this event happens when a target is detected, which is achieved through collision shapes. These mimic a sort of 'vision range' for all units, which makes them aware of others inside of that range. Even though realistically a cone would make more sense, in the base implementation a circle is chosen for computational and implementational convenience. For both ally and enemy units, these circles can be visualized as shown in Figure 5.1.

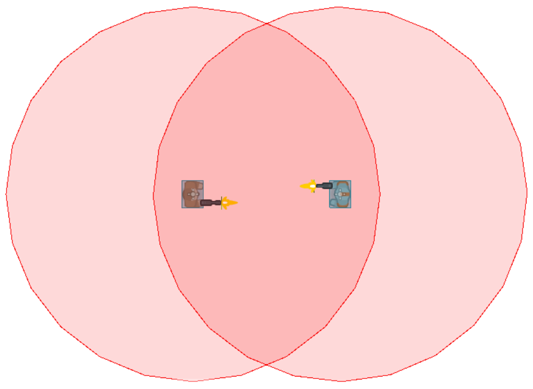


Figure 5.1: Collision shapes visualized in red for ally and enemy units.

5.1.1 Problem

As mentioned above, only having one-on-one combat is not quite realistic. In a real-life scenario, other friendly units would be notified about the encounter (either verbally, through digital communication, or by hearing gun sounds) and start to engage themselves. Hence, as a first basic communication strategy, a teammate-notification protocol is introduced. For this, the following two parameters from Section 4.3.1 are implemented into the game code:

- **communication_count**
  The number of teammates to notify when a unit engages in combat.

default value	**minimum**	**maximum**	**step size**
2	0	5	1

Table 5.1: communication_count parameter attributes.

- **communication_delay**
  The delay between a unit starting a combat encounter and sending out the notification to the unit's teammates.

default value	**minimum**	**maximum**	**step size**
0.2	0.0	3.0	0.5

Table 5.2: communication_delay parameter attributes.

Once a unit engages an enemy, it notifies the others through a higher-level communication class. This class awaits the delay, after which it collects all remaining teammates and sorts them based on the distance to the already attacking unit. The first *communication_count* of these are signalled to attack the opposing team's unit, bypassing the need for spotting the enemy in their vision range.

As a first implementation of this signal behaviour, the units that get notified just start shooting in the direction they're told, without checking whether they will be hitting the enemy or any in-between obstacles. This is a naive implementation, so the expected outcome of the optimizer is to showcase the shortcomings thereof. As a scoring metric, the amount of team damage is chosen (as the simple implementation will negatively impact this value). Logically, one would expect that adding communication and teamwork would improve unit cooperation and hence positively influence the game outcome.

5.1.2 Optimization

Setting the initial parameter value for the communication count to 2 and the initial value for the communication delay to 0.2, the graph in Figure 5.2 is produced. The x-axis shows the iteration number, the y-axis shows the average ally team damage per instance, and each data point has a label that shows the input parameters $(communication\_count, communication\_delay)$. Intuition would say that a higher number of notified units is a better strategy. However, on the graph, it seems like the optimizer settles for a low communication count of 0 (paired with a trivial communication delay, since it has no effect if there are no units to communicate with). The average team damage dealt is below 20 for all iterations where the count is set to 0, which means that on average less than one bullet fired by an ally hits another allied unit (as bullet damage is equal to 20).

The data points can also be plotted on a heatmap as can be seen in Figure 5.3. To ensure readability, each cell represents not just one but several experiments, depending on how many $(count, delay)$ values are within a bucket range of 0.00001 of the actual cell coordinate (e.g. for a cell $(1, 0.00020)$ all values from $(1, 0.00019)$ to $(1, 0.00021)$ are collected). The experiment results are averaged over this neighbourhood, as the optimizer sometimes explores a narrow area around certain points.

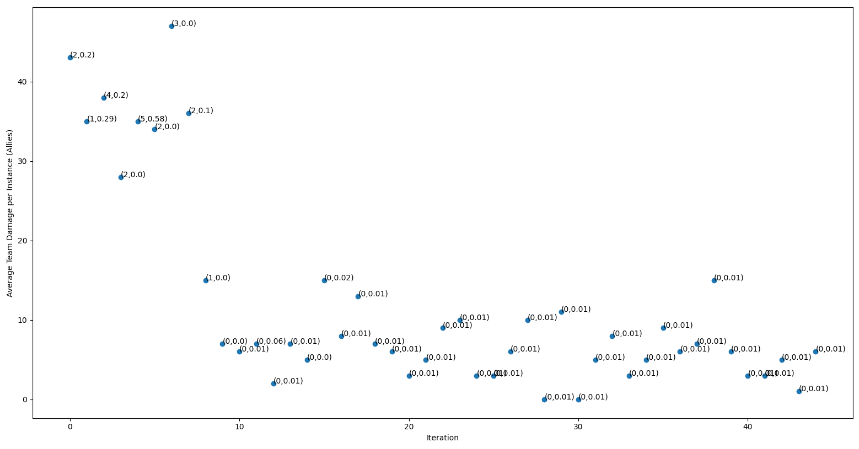


Figure 5.2: Notify-on-attack experiment results. Points labelled with (communication_count, communication_delay).

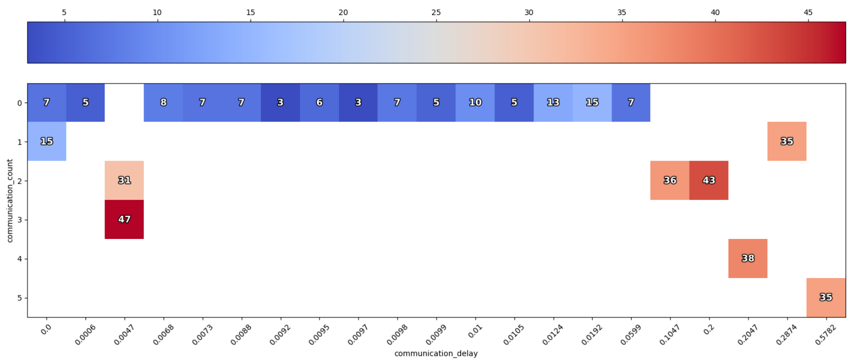


Figure 5.3: Notify-on-attack heatmap. Points labelled with average score of that (delay, count) parameter pair. A lower score is a better result.

However, it is also possible that the algorithm finds a different optimum based on the given bounds for the parameters. For example, if the upper and lower bounds for the communication delay get changed to $[0, 5]$, the optimizer can be persuaded to look at a high communication count value. This happens because a delay which is high enough is practically equivalent to having a communication count of 0 (as by the time other units get signalled, the attacking and/or attacked unit are already dead). An example of such an optimization run can be seen in Figure 5.4, with a heatmap showing the overly explored communication count value of 4 in Figure 5.5. This shows how important it is to always consider the upper and lower bounds of parameters and how they interact when evaluating an experiment.

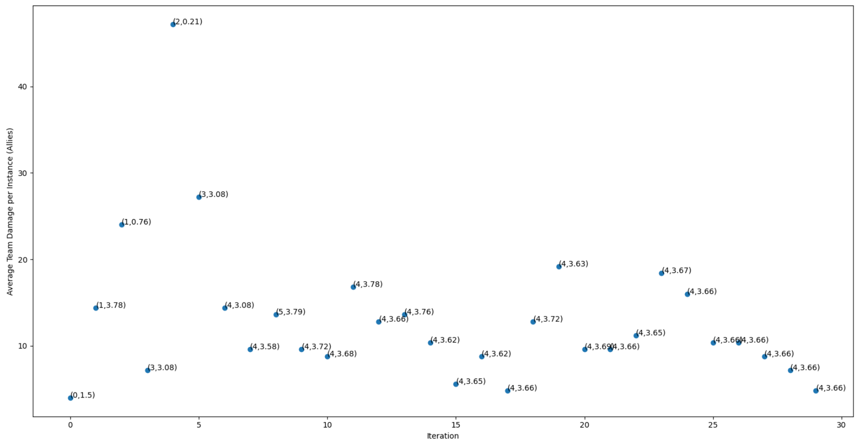


Figure 5.4: Notify-on-attack experiment results with alternative parameter bounds. Points labelled with (communication_count, communication_delay).

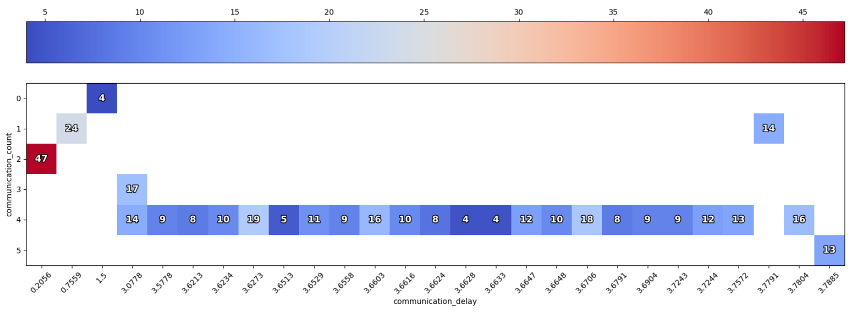


Figure 5.5: Notify-on-attack with alternative parameter bounds heatmap. Points labelled with the average score of that (delay, count) parameter pair. A lower score is a better result.

The experiment has shown that, in the current implementation, a higher communication count negatively impacts team damage. This can be explained by the initial setup of the ally and enemy units, as they follow a different movement pattern; the allies move from left to right in single file formation, while the enemies patrol the area independent of one another. Since notification happens in a circular manner around the engaging unit, if the closest units are right behind one another they won't have the room to aim towards the attacked unit properly. As can be seen in Figure 5.6, this is the case for the allies, whilst the enemies can engage in a cross-fire pattern (as they are more spread out).

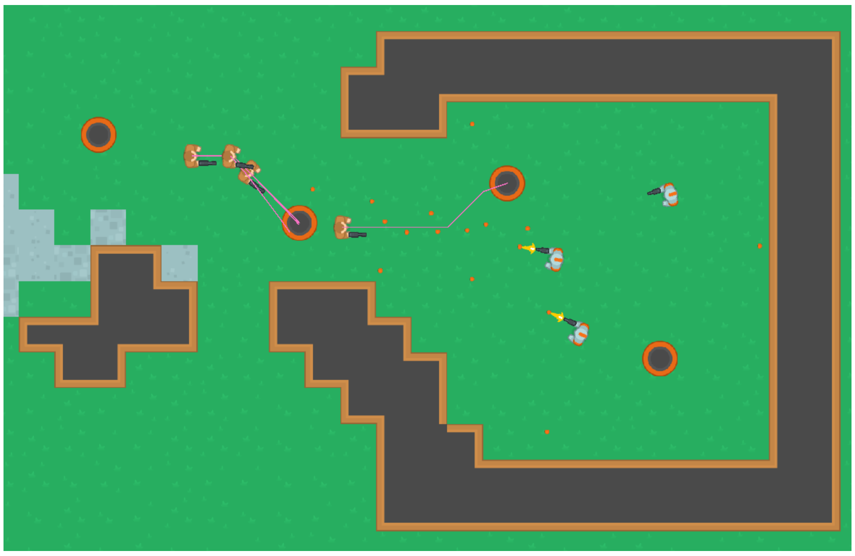


Figure 5.6: Cross-fire behaviour of enemy units.

5.2 Alternative Scoring Metric

In the previous experiment, the only element of the state that was taken into account for scoring an optimization iteration was the team damage done by the allies. However, this is not the only aspect worthy of optimizing. An alternative scoring metric that not only values low team damage but also incentivizes winning the game can be used to analyse the basic notification protocol from the previous experiment.

5.2.1 Problem

Evaluating the game on multiple aspects can show how dynamic the current version truly is. By implementing an alternative scoring metric that weights values related to winning the game (which is achieved by having killed all enemy units) this can be achieved. A code snippet can be seen in Figure 5.7, where the score is computed from a given game instance state (see Appendix A.1 for an example of such a state file) by giving a high penalty of 1000 for losing all allies, a small penalty of 10 for all remaining enemies and a large bonus of -500 if all enemies are killed. The amount of team damage is still used but has less impact on the overall score.

```python
def score_game(self, game_results:dict) -> float:
    score:float = 0.0
    alliesAlive = 0
    for unit in game_results["allies"].values():
        if unit["health"] > 0:
            alliesAlive += 1
    if alliesAlive == 0:
        score += 1000 # high penalty for losing all allies

    enemiesAlive = 0
    for unit in game_results["enemies"].values():
        if unit["health"] > 0:
            enemiesAlive += 1
    if enemiesAlive != 0:
        score += 10 * enemiesAlive # weighted penalty for keeping more enemies alive
    else:
        score -= 500 # bonus for killing all enemies

    score += game_results["team_damage"]["allies"] # penalty for team-damaging allies
    return score
```

Figure 5.7: Python code for alternative game scoring using weighted metrics.

5.2.2 Optimization

This time the naive approach of the notification protocol shows its shortcomings. Intuitively it would make sense to have more friendly units helping in combat, but the results as seen on the graph in Figure 5.8 as well as the heatmap in Figure 5.9 indicate that the best value for the communication count is still 0. Similar to the previous experiment, this can be explained by the movement patterns of both allies and enemies. Since the single-file movement of the allies does not benefit from the notification protocol, whilst the cross-fire potential of the enemy patrol positioning does, a higher communication count will only be an advantage to these enemy units.

This shows that some better implementation is required to make the behaviour of the units feel more realistic, since the current outcome does not match the intuition that teamwork is better than one-on-one combat engagement.

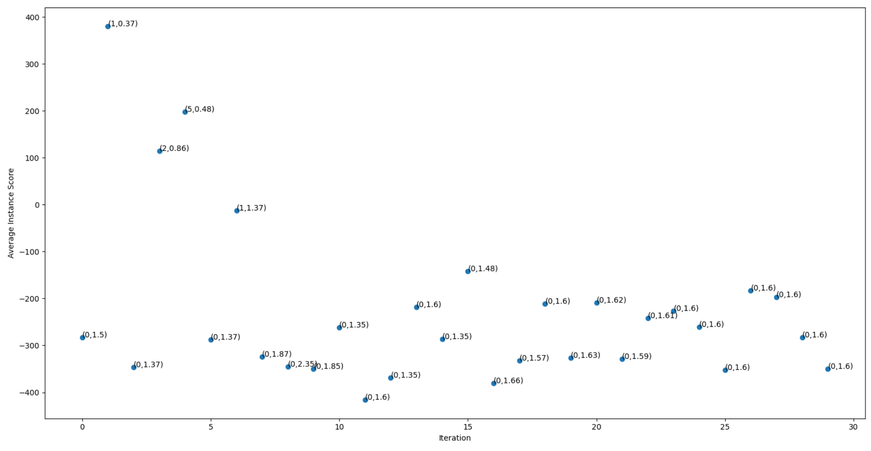


Figure 5.8: Notify-on-attack experiment results with weighted scoring metric. Points labelled with (communication_count, communication_delay).

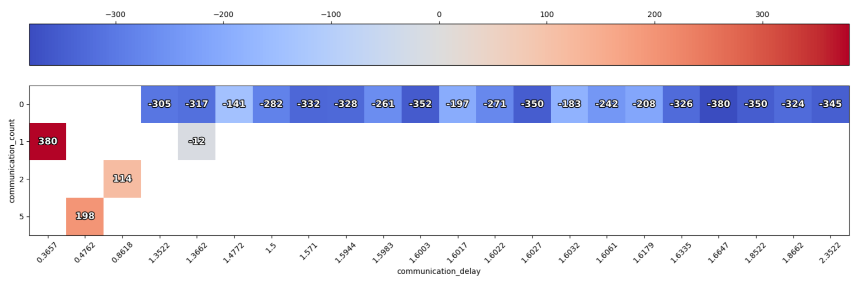


Figure 5.9: Notify-on-attack with weighted scoring metric heatmap. Points labelled with an average score of that (delay, count) parameter pair. A lower score is a better result.

5.3 Improved Communication Strategy

Since both prior experiments have shown that the basic implementation of the communication strategy is not quite perfect, a revisit of the 'notify' strategy is in order. In addition to the point-and-shoot mechanic, tactical movement can be introduced to also reposition the notified friendly units.

5.3.1 Problem

The previous experiment shows how the initial implementation of a communication strategy does not quite match expectations. Hence, the 'notify' strategy should be revisited so units react smarter when they are tasked with aiding their teammates. In the old implementation, when a unit starts attacking, the notification protocol selects the *communication_count* closest friendly units and informs them about the location of the opponent that the attacking unit is targeting. This has some issues since it is not certain that the closest units can actually see this opponent. Instead of this simple point-and-shoot implementation, tactical movement can be introduced to guide units to better positions when engaging with the enemy. Now, when a friendly unit starts attacking someone within their vision range, the first *communication_count* teammates will reposition themselves in such a way that they can shoot (and hit) the common target.

5.3.2 Optimization

As introduced above, the new communication strategy to optimize uses tactical movement to reposition friendly units. This can be visualized as shown in Figures 5.10 and 5.11, where the two grey enemy units in the bottom right move towards the position of their under-attack friend in the middle. The map layout has also been updated to show this tactical movement better since adding an obstacle reduces the number of vision lines between units. The same parameters can still be optimized for this experiment, being the *communication_count* and *communication_delay*. As a scoring metric, the weighted version from Section 5.2 is used.

When looking at the results in Figures 5.12 and 5.13, an optimum is no longer found at a count of zero, but the optimizer finds that a count of one is best. Surprisingly, a high number of notified units is not good either, which can be explained by the fact that both the allies and the enemies use the same notification-reposition protocol. Since the general setup of both teams' initial behaviour remains largely unchanged, the enemies still have a slight advantage in engaging in cross-fire when a high communication count is selected. Still, these results showcase a sort of emergent buddy tactic by setting the count to one, which works better for the allied units. Nevertheless, en masse communication benefits the enemies more, as it allows them to flank allied units from multiple sides since each enemy patrols a different area before engaging.

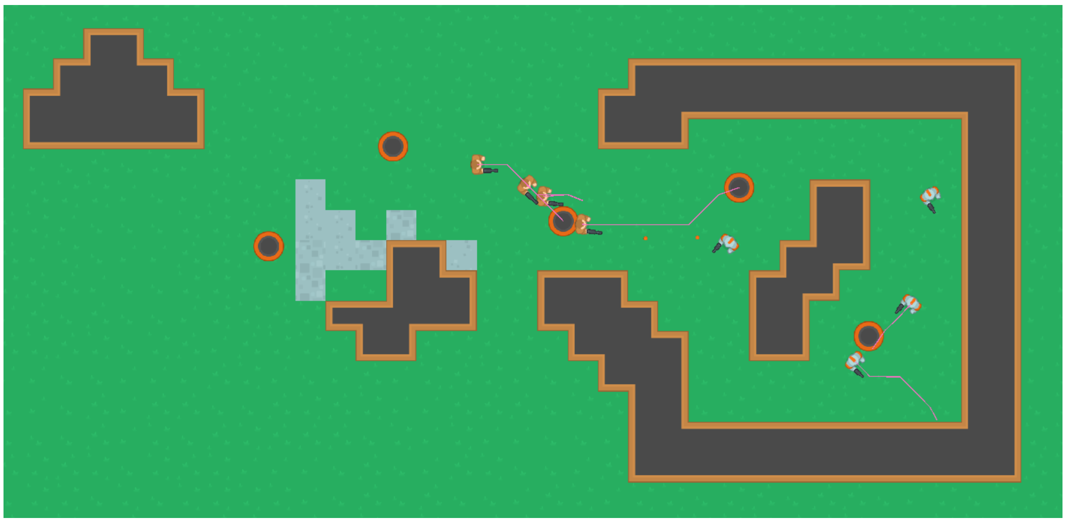


Figure 5.10: Visualized opponent engagement before notification of friendly units.

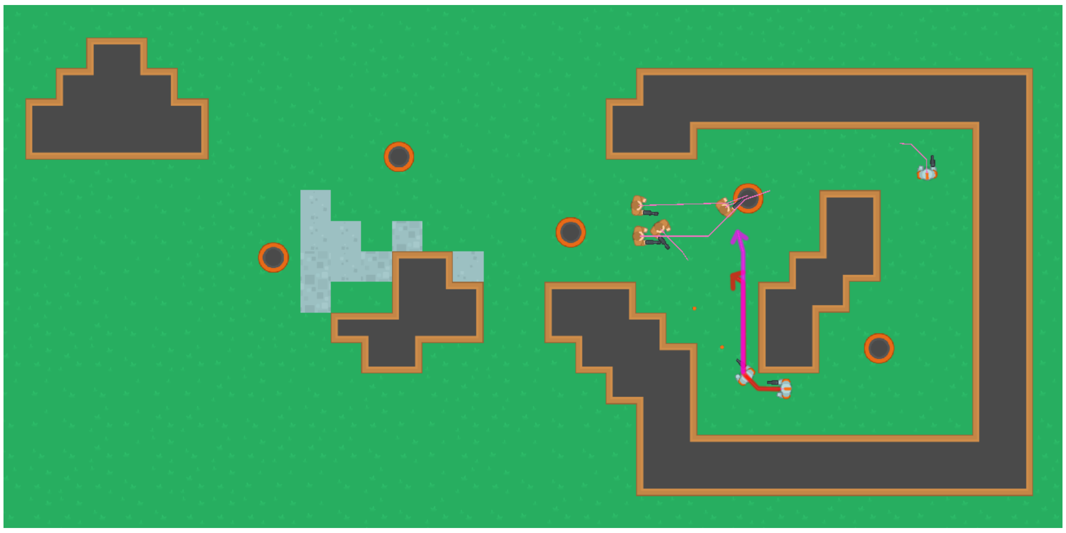


Figure 5.11: Visualized tactical movement of lower two enemy units.

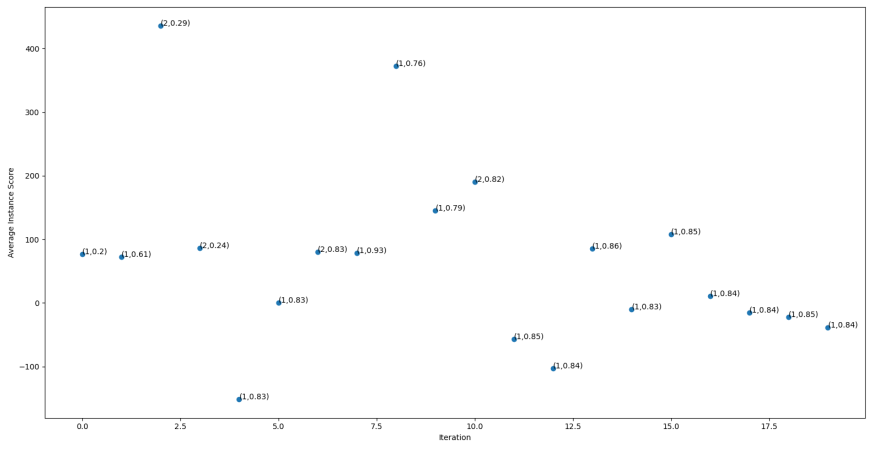


Figure 5.12: Improved communication protocol experiment results. Points labelled with (communication_count, communication_delay).

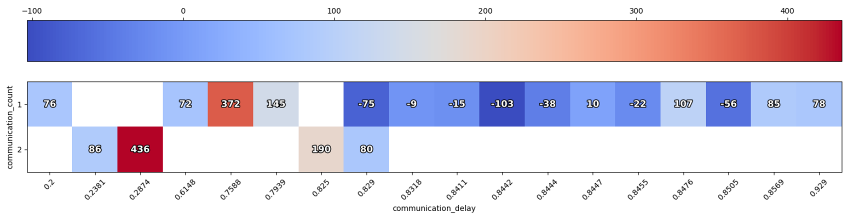


Figure 5.13: Improved communication protocol heatmap. Points labelled with an average score of that (delay, count) parameter pair. A lower score is a better result.

5.4 Vision Cone

As shown in Figure 5.1, the current way for units to notice one another is by checking collision with a circular area around themselves. If another unit B's geometry intersects with the area of a unit A, and B is of the opposing team, A can start attacking this invading opponent. This simplified implementation does not feel realistic, as when one unit is looking in the opposite direction, it has the same vision circle as an approaching opponent. They both 'spot' each other at the same time, whilst the tactical advantage should go to the stealthily approaching unit. The following experiment explores the potential alternatives, how they are implemented, and an optimization run with the Framework to determine what vision shape works best. To avoid side effects, the communication protocol from the previous experiments has been disabled. In terms of the scoring metrics, the weighted scoring is re-used.

5.4.1 Problem

Humans do not have 360-degree vision, but rather a visual field of about 200-220 degrees. Within this area, the inner 120 degrees are part of our binocular vision range (where both eyes can see objects, allowing for depth perception) [72]. Inside this region, there is a slice of about 60 degrees which is the actual focus range for symbol recognition [73] and an even smaller wedge of 30 degrees which is the preferred viewing area [74]. To mimic realistic perception for the in-game units, similar values should be used (instead of a circular 360-degree viewing area).

Alongside the viewing angle, a viewing distance parameter should be added. In the game setting, this does not need to be as realistic as the angle is. This is because NPCs that can look 'infinitely far' (like humans can in real life) would make for a dull experience. When introducing a player character, they would feel limited by screen resolution and map rendering whilst the NPCs can see one another (and the player) from across the map. Hence, the viewing distance should be limited to a reasonable value.

5.4.2 Optimization

This time, a very different optimization is needed in contrast to the last three experiments. Thus, both the Optimizer as well as the game itself should be updated to accommodate the newly designed vision cone feature. The easiest of the two to adapt is the optimizer since it was designed with reusability in mind. A new parameter class is introduced, being the *VisionConeParameters*. The Python code for the class definition is shown in Figure 5.14, wherein both the vision distance and angle are defined with their respective upper bounds, lower bounds, initial values and step sizes.

```
from Parameters import GenericParameters, GenericParameter
class VisionConeParameters(GenericParameters):
    def __init__(self):
        self.vision_distance = GenericParameter(type_="float", min_=100, max_=600,
                                                value=300, step_size=50)
        self.vision_angle = GenericParameter(type_="float", min_=10, max_=220,
                                                value=60, step_size=15)
```

Figure 5.14: Python code showing an example GameParameters class definition.

To accommodate using this new parameter set instead of the old *GameParameters*, another derived class can be made. This time, most of the necessary features already exist in the *GameOptimizer* class. Hence, a minor update to that class is enough to allow for inheritance and using the new parameters. This relation is visualized in Figure 5.15, where the *set_parameters()* method is used to set the class instance's *self.parameters* value to be an instantiation of the *VisionConeParameters* class. The updated *run_subprocess()* function is used to pass these parameters to the game executable, by providing both *vision_distance* and *vision_angle* parameter values as command line arguments. Since the same scoring metric from prior experiments is used, no update to the *score()* or *score_game()* methods is required. The same holds for all the data plotting and storing helper functions, as well as the initialization, objective, and run methods.

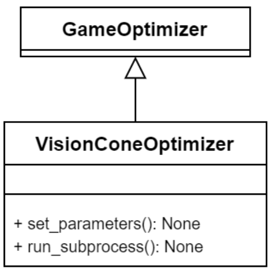


Figure 5.15: Class diagram of the Vision Cone Optimizer (inheriting from the Game Optimizer).

On top of the Optimizer updates, the actual vision cone behaviour should be implemented into the game itself. For this, two potential candidates come up; the first being the simplest, namely drawing an isosceles triangle that has a height equal to the vision distance, and a top angle being the given vision angle. Alternatively, the vision cone could be drawn as a circle slice which has the vision distance as a radius, and the vision angle as the maximal sector angle to draw. Both approaches are shown side by side in Figure 5.16[1].

[1]An interactive visualizer is available at `https://joshuamoelans.github.io/thesis/AngleGen/`

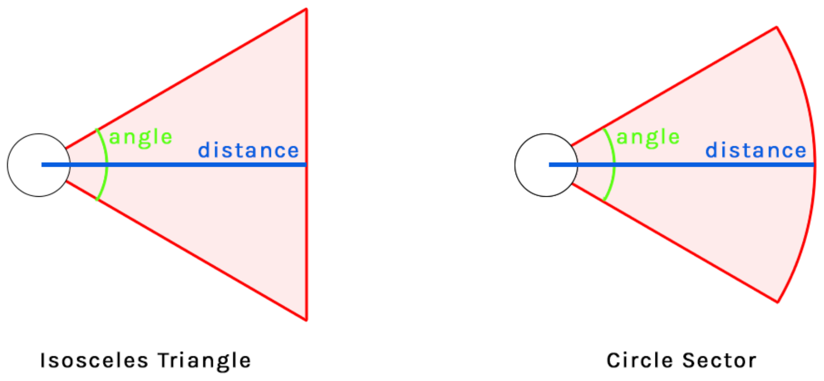


Figure 5.16: Comparison of two approaches to the Vision Cone implementation.

The first implementation has some issues since the top and bottom coordinates of the triangle need a y-position. One way to calculate it is by using trigonometry since the distance (being the height of the triangle) and the top angle of the triangle are given. The y-position comes out to $y = \pm\tan(angle/2) \cdot distance$, where the sign indicates the point being above or below the view horizon. The problem is that the tangent function is asymptotic on multiples of 90 degrees. In these situations, the y coordinate would become infinity (leading to an extremely wide triangle). To solve this, the alternative method was implemented using circle sectors instead. These nicely cut off the vision cone such that any visible point is at most $vision\_distance$ away from the observer. A code snippet of the GDScript implementation of this is shown in Figure 5.17, where given an angle, distance and resolution, a circle sector is constructed by generating a polygonal vision cone. The radius of the circle is equal to the distance, and the angle decides the cut-off of the circle slice. The given resolution is used to determine how many triangles are drawn as part of the polygonal sector.

When removing the circular vision area, the ability for units to see behind them was removed as well. However, a feature is now needed to notify any unit under attack of the direction they're being attacked from. Otherwise, they could potentially keep their back turned towards the danger. For this, a *ROTATE* state was added to the behaviours of the units. Once they detect that they are under attack (by getting hit with a bullet), they will rotate towards the direction they're being hit from, rotating their vision cone with them. This way, there is no need for a circular 'notification' area to inform units of the opponents' presence.

```
func setCone(angle, distance, resolution=30):
    angle = angle * PI / 180
    var half_angle = angle / 2
    var step = angle / resolution

    var points = PackedVector2Array()
    points.append(Vector2(0, 0))

    for i in range(resolution + 1):
        var theta = -half_angle + step * i
        var x = distance * cos(theta)
        var y = distance * sin(theta)
        points.append(Vector2(x, y))

    points.append(Vector2(0, 0))
    visionCone.polygon = points
```

Figure 5.17: GDScript code showing the circle sector vision cone computation.

With both the Optimizer and the game ready for action, an instance of the Optimizer can be run. The distance and angle parameters are set up with default values, upper and lower bounds and a step size as shown in Tables 5.3 and 5.4. Running the *VisionConeOptimizer* for 40 iterations, the results as shown in Figure 5.18 and those in the heatmap in Figure 5.19 are produced.

- **vision_distance**
  How far the vision range of a unit is.

default value	**minimum**	**maximum**	**step size**
300	100	600	50

Table 5.3: vision_distance parameter attributes.

- **vision_angle**
  How wide the field of view is for a unit.

default value	**minimum**	**maximum**	**step size**
60	10	220	15

Table 5.4: vision_angle parameter attributes.

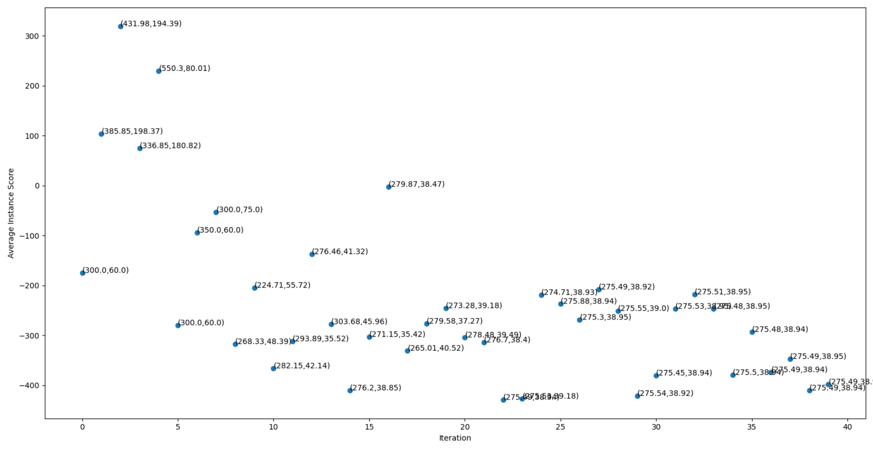


Figure 5.18: Vision cone experiment results. Points labelled with (vision_distance, vision_angle).

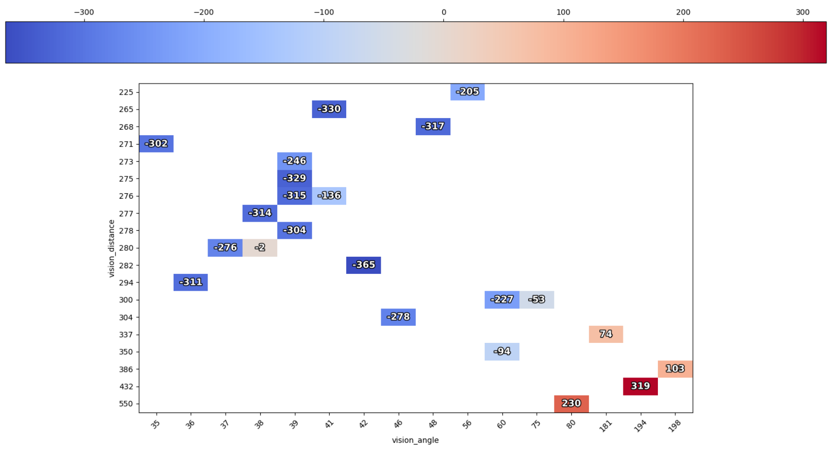


Figure 5.19: Vision cone heatmap. Points labelled with an average score of that (angle, distance) parameter pair. A lower score is a better result.

The optimizer's multi-start MLSL algorithm explores a wide range of values, within the given minima and maxima. The vision angle is explored between 35 and 198 degrees, whilst most of the vision distance range is also evaluated, seeing values between 225 and 550 matched with a variety of angle values. A clear difference in the results can be seen in the heatmap, where a cut-off point for the vision angle is found around the 60-75 degree point. All values with a wider angle have worse results than those within the 35-60 range.

A similar correlation occurs for the vision distance, albeit less noticeable. This can be explained by the fact that this experiment reuses the updated map from the **improved communication protocol** experiment in Section 5.3 (where an additional obstacle was added, hence seeing much further does not help the allies). Also, since the allied units attempt to breach the protected' area on the right, it makes sense that for them to score better, it is best that the enemies can not see into the far distance (since this makes them aware of the line-up of allies coming their way).

Thus, it seems like the optimizer finds angle values within the realistic focus range of 30-60 degrees to be the best-performing ones. The introduction of this experiment documented how humans can focus on symbols at angles less than 60 degrees, and the preferred viewing area resides anywhere from 10-30 degrees, which is matched by the optimizer results. In terms of vision distance, a shorter range is preferred to give the allies an upper hand.

CHAPTER 6

Conclusions and Future Work

In this thesis, automated parameter optimization for behaviour space exploration was researched. This was motivated by the large cost of both time and money that gets spent on playtesting when developing a game. To act as a realistic showcase, a game was developed in a commercially viable engine (the Godot game engine). A generic optimization framework was presented as an input-output system, which uses NLopt's suite of optimization algorithms. Next to the generic framework, instructions for translating it into a concrete class for optimizing a video game are given. Several experiments emulating an iterative feature-finetune cycle were done, each providing valuable insights into new or existing in-game functionality and behaviour.

The main contributions are described once more in Section 6.1, with additional contextualization given the contents of the research done. Lastly, some future work is described in Section 6.2

6.1 Contributions

Given the contents of all prior Chapters, another look at the main contributions can now contextualize them within the scope of the thesis.

6.1.1 Optimization Framework

Chapter 4 presents both the generic framework as well as the concrete implementation needed to optimize a video game. Additionally, an abstract and concrete parameter class is designed to simplify the process of giving the optimizer upper and lower bounds, initial values, initial step sizes and parameter types.

To showcase how this framework operates in the video game development setting, a simple tactical top-down shooter game is built in Chapter 3 by using the commercial Godot Engine. During the design process, some thought is given to providing the optimizer framework with an easy way to input parameter values, as well as a method to extract output by saving the full game state. To reduce variance in the experiment output, the game is also prepared for in-game and across-game instancing. This is achieved by replicating a single instance of the game world into a grid (running in one executable) and running multiple of these grids in separate windows.

The additional workload for adding the hooks needed for optimization into an existing game is negligible. The input parameter values can just be passed to the executable by command line arguments and parsed into their proper scripts by trickling down through the game's code. Since most games have some method of saving data already, adding the hook for saving the output (being the state) of the game into a file is a minimal effort as well.

6.1.2 Proof-of-Concept Experiments

In Chapter 5 both elements of the Optimization Framework as well as the example video game are used to analyze a workflow combining the power of optimization with large-scale game simulation. This methodology intertwines both feature implementation with automated parameter optimization, intending to find values for features within the realm of tactical communication and character behaviour.

An iterative procedure is followed where the game is analyzed as-is, and then a suggested new feature or improvement on an existing feature is proposed. By implementing this feature into the game, setting up the framework's optimization parameters and performing a full optimization cycle, interesting results are produced which can help inform decisions in the game development process.

This thesis explores a basic communication strategy between military units, of which a first implementation is shown to be incomplete by the optimizer (as the results do not match intuition). By improving the implemented notification protocol, a second run of the optimizer results in an emergent buddy tactic which is more realistic. Lastly, an experiment is performed on NPC vision cones, where the results match the biology of the preferred viewing area for human vision.

6.2 Future Work

In this Section, some potential future work related to the thesis is laid out. These range from game-specific ideas (such as improved pathfinding algorithms, alternative features and their accompanying Optimizer experiments) as well as game-agnostic concepts (such as generating code from behaviour statecharts, using scoring metrics to set game difficulty or even learning player movement).

6.2.1 Using Metrics for Game Difficulty

As seen in Chapter 5 different scoring metrics can lead to vastly different results in the Optimizer. An example of a simple metric using one value extracted from a game was shown, as well as a more involved weighted scoring that was a measure of how often the allied units win. From this second one, a potential research avenue can be derived.

Games often offer multiple difficulty levels to make it easier or harder for the player to win. During the game's development, a lot of time can be spent not only optimizing values for a single result but also making sure that the game feels appropriately demanding for the selected difficulty level. Here the proposed Optimizer can be employed, for example by setting a threshold on the average win or lose percentage depending on the selected difficulty. That way, automatically finding good parameter values for each of the difficulty levels can help make the game feel more balanced without requiring extensive manual fine-tuning.

A similar concept has been explored by Robin Lievrouw in their thesis on *Applying Dynamic Game Difficulty Adjustment Using Deep Reinforcement Learning* [75]. It differs from the future work concept this thesis puts forth by having the difficulty adapt based on the actual player playing the game, and not by pre-learning set difficulties through parameter optimization. The results from the research show that for humans the changes are too minute to lead to any mood change, implying that the system might be overkill for what it tries to achieve. Finding a middle ground between general difficulty learning by parameter optimization and the dynamic difficulty adjustment idea seems like an interesting area of research.

6.2.2 Statechart Code Generation

In Chapter 3 Section 3.2.4 the behaviour of in-game units is given entirely in terms of a statechart. In implementation, this is then translated manually into GDScript code specific to the Godot Engine. However, this statechart could be reused for alternate code generations to other game engines such as Unity [56] or Unreal [55], or even JavaScript frameworks like Phaser [76] or PixiJS

[77]. If the same input-output mechanisms are provided in the alternative game implementations, these too could slot into the framework for automated optimization.

It would also be possible to use a specific engine/framework to do the optimization loop, whilst implementing the full game (with the learned parameters embedded into the statechart) in another engine. This can be useful when a game requires high graphical fidelity upon release (which can more easily be achieved in Unreal Engine) but the learning can happen at a lower resolution' engine like Godot.

This concept is a similar yet toned-down version of the research done by Togelius and Schmidhuber in their paper *An Experiment in Automatic Game Design* [78]. They propose generating a game from a set of rules (rather than behaviours from a statechart) which, given the vast space of even a constrained set of game states, gives way for millions of potential results. Not all of these are necessarily playable, yet they can serve as exploratory prototypes which might turn out to create some emergent new game genre.

6.2.3 Cooperative Pathfinding

As mentioned in Section 3.2.5 multi-agent pathfinding is a hard problem to solve. In the thesis game implementation, this was largely ignored to avoid overcomplicating this small aspect of the game. However, some techniques have been proposed that aim to implement a smarter' pathfinding that can run dynamically. David Silver's paper on WHCA* [79] is an example of such an algorithm, which uses three-dimensional coordinates to find paths on a two-dimensional grid. It is still based on the A* algorithm but uses non-colliding routes and a windowed hierarchical search.

The technique proposes adding a dimension t to the (x, y) coordinate grid. This dimension represents the time aspect, such that any agent that plans out a path from point (x_a, y_a) to (x_b, y_b) reserves the in-between positions (x_i, y_i) at the estimated time i (and potentially a buffer ϵ around the timestamp i) such that they might pass through that coordinate marking it as impassable for any other agents. This search is done in so-called search windows' to limit the number of computations needed. This way, an agent always computes a partial route towards their end goal and recomputes at either fixed timestamps (offset with other agents to lower peak computation) or after a certain distance is travelled.

6.2.4 Learning Player Movement

Until now the only optimization that was done happened on the AI of in-game NPCs. It might however be interesting to allow for parameterizing player characters as well, optimizing both their variable values as well as the interaction with NPC systems. Simple player behaviour can be programmed by taking NPC behaviour and retrofitting it into autonomous player controls. However, nowadays it is possible to learn player movement through Neural Networks. Similar to Google Deepmind's AlphaGo [12], a network can be trained on both self-play and collected human experiences (for example from a limited amount of playtesters). Creating an artificial player this way can provide a better groundwork for parameter optimization, as often there is a certain nuance to how people play games as opposed to a manually implemented simulation thereof. Balancing the self-learning and experience-learning aspects of the network training phase is critical since the artificial player should not feel too artificial (and keep using a lot of the collected human experience). This approach can help the framework build an artificial dummy human to also optimize parameters for their in-game player character, as well as learn parameters of the auxiliary systems that interact with them.

6.2.5 Additional Optimizer Experiments

For the specific game that was built for this thesis, some additional experiments were designed. This Section briefly outlines some of the ideas, and why they are interesting features to investigate.

Enemy Priority

Currently, all units have an 'enemy buffer' queue which tells them the target priority of enemies. However, no feature smartly decides what these priorities should be. Instead of relying on the first-come-first-served system as it is now, it would make sense to assign weights to the spotted opponents to be able to reassign priorities dynamically. For example, if a unit knows of an enemy's position, it might not make sense to attack that enemy first if their nearby friends are already attacking someone else.

In terms of parameters to optimize, this feature could introduce some weights that act as attention dividers; how much does being nearby influence the necessity of a reaction? Is it better to attack in a group or do the results indicate that units should attack separately? Running the Optimizer and letting it play hundreds of games can give an insight into what weights positively influence the chances of winning.

Line-of-Sight Avoidance

On top of the cooperative pathfinding concept from Section 6.2.3 there are still some other issues with unit movement specific to the game designed for this thesis. In general solving movement by avoiding intersecting paths is a great first step, but in a shooter scenario, it would make sense that the line of fire of friendly units is avoided. This was explored in a research internship by Barmon in 2018, where coordination to reduce the impact of friendly fire was introduced [80].

The proposed solution involves an adaptation to the A* pathfinding algorithm, where a restricted area is drawn in front of any ranged unit which heavily increases the weight of the positions within this area. When another unit tries to find a path to a position, this path will avoid the restricted area as much as possible (since it is better to cross when under attack, so not crossing could be worse). Additionally, the idea of target selection is presented. This ties in with the previous experiment idea of **Enemy Priority**, since friendly fire can be reduced by ensuring enemies are targetted at appropriate times, avoiding intersecting paths with active lines of fire.

To implement this in-game, the vision cone from the experiment in Section 5.4 as shown in Figure 6.1 can be reused. The area that the cone overlaps should be marked as terrain that should not be traversed, which is achieved by assigning higher weights to the pathfinding grid cells. This can be visualized as red-coloured cells, indicating a non-zero weight in that area. A potential parameter set to optimize would then be the actual weights that are assigned to each grid cell and how this influences the amount of friendly fire. In contrast to the prior research by Barmon, a non-uniform weight could be used to further discourage traversing certain areas of the vision cone.

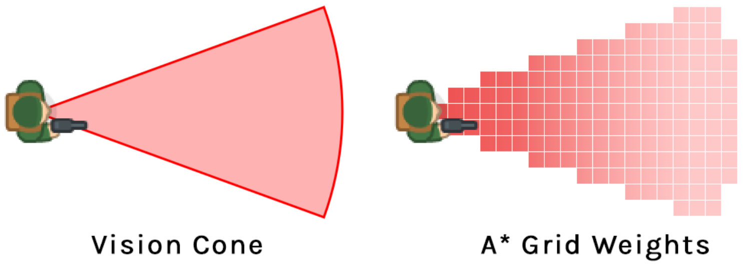


Figure 6.1: Visualization of vision cone and accompanying A* grid weights for line-of-sight avoidance.

Getting Attacked Response

In the first implementation whenever a unit got attacked the response was implicit by nature of the circular vision area around each unit. They would either see the opponent at the same time that they got spotted, or when getting hit a minor increase in vision area was done to avoid needing pixel-perfect overlap (simulating a sort of heightened awareness). The final experiment in Section 5.4 improves upon this behaviour by changing the vision range to a conical shape. When a unit gets hit now, it turns towards the direction of the bullet. This is already an improved behaviour, yet it is not entirely natural; the unit under attack starts attacking the opponent but has no behaviour to strategically position themselves or to try and shoot back whilst running away.

The feature to implement and optimize would be a tactical repositioning away from the bullet direction. This can for example be done via spline movement where a unit is instructed to move from A to B via in-between point C to get to a tactical position (e.g., closer to the opponent or closer to their friends). An example is shown in Figure 6.2 where an incoming attack vector from the right results in a reposition spline going around the incoming attack line. The parameters for this could be the maximal/minimal movement distance, how curvy the spline path should be or even the decision mechanism that makes a unit stay in their spot (for example when they have more health than their squad mates) or run away.

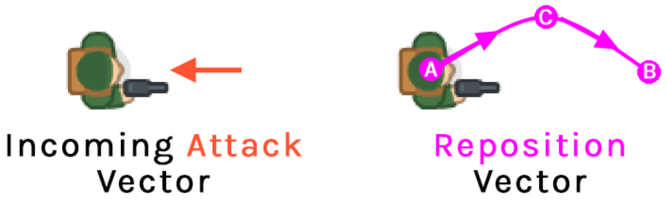


Figure 6.2: Visualization of incoming attack vector and resulting tactical reposition vector.

Squadron Movement

In the current game implementation a squadron of units moving from A to B line up perfectly, following neatly behind one another. This is not quite realistic, as the so-called ‘column’ or ‘single-file’ movement is reserved for routes where enemy contact is not expected [81]. Alternative patterns such as Wedge, Staggered Column, Vee or Line make more sense in a scenario where an active threat is present, which is the case in the game environment designed for this thesis. A visual comparison of these formations is shown in figure 6.3.

Hence, as a potential next feature, simple squadron movement could be added. These can each be parameterized on their own (e.g., how much distance between units, how far they should be spread, maximal angles between units, ...) to find an optimum for a specific pattern, or they can be selected through the use of an enumerator-like parameter, which then evaluates how good each formation is within the game setting.

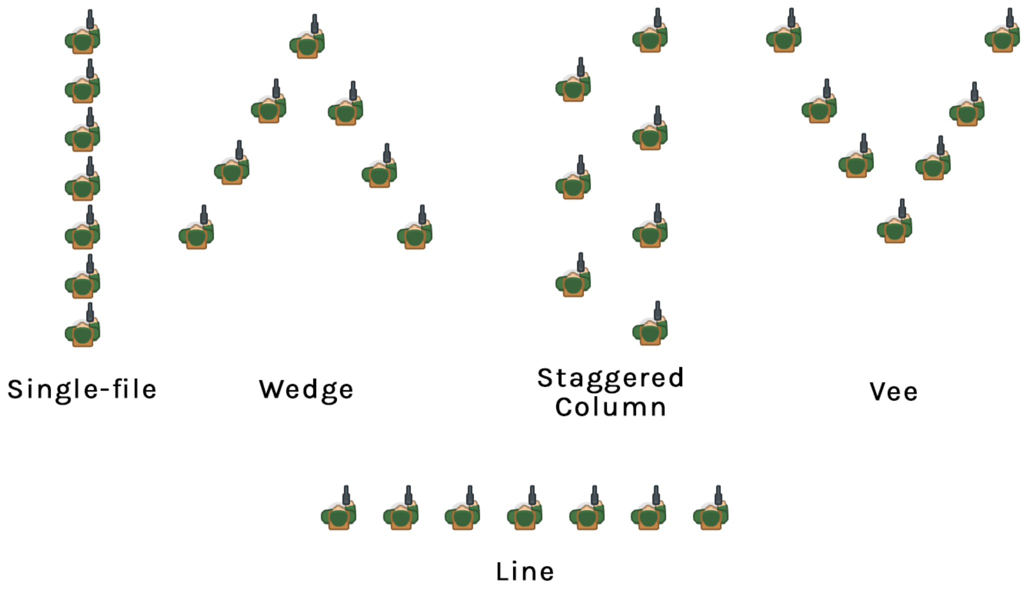


Figure 6.3: Visualization of different tactical formations for squadron movement.

Bibliography

[1] R. Verschelde, "Engines/frameworks used in Global Game Jam," Accessed on 2024-04-13. [Online]. Available: https://twitter.com/Akien/status/1751904118132683081

[2] K. Vanherpen, J. Denil, P. De Meulenaere, and H. Vangheluwe, "Design-space exploration in model driven engineering," in *SOCS-TR-2014.4*. McGill University, 2014.

[3] L. M. Zintgraf, L. Feng, M. Igl, K. Hartikainen, K. Hofmann, and S. Whiteson, "Exploration in approximate hyper-state space for meta reinforcement learning," *CoRR*, vol. abs/2010.01062, 2020. [Online]. Available: https://arxiv.org/abs/2010.01062

[4] G. Hackenberg and D. Bytschkow, "Towards Early Emergent Property Understanding," in *Proceedings of the 1st Extreme Modeling Workshop at MODELS*, 2012.

[5] J. Gomes, P. Mariano, and A. L. Christensen, "Systematic Derivation of Behaviour Characterisations in Evolutionary Robotics," *arXiv preprint arXiv:1407.0577*, 2014.

[6] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, 1987.

[7] S. Bromley, "How to budget for games user research," 2023, Accessed on 2024-03-28. [Online]. Available: https://gamesuserresearch.com/how-to-budget-for-games-user-research/

[8] E. Crichton-Stuart, “Indie games claim 31% of all steam revenue,” March 10 2024, Accessed on 2024-03-28. [Online]. Available: https://gam3s.gg/news/indie-games-claim-31-perfect-of-all-steam-revenue/

[9] S. G. Johnson, “The NLopt nonlinear-optimization package,” https://github.com/stevengj/nlopt, 2007, Accessed on 2024-03-24.

[10] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Ferndez del Ro, M. Wiebe, P. Peterson, P. Grard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, “Array programming with NumPy,” *Nature*, vol. 585, pp. 357–362, 2020.

[11] J. D. Hunter, “Matplotlib: A 2d graphics environment,” *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, “Mastering the game of go with deep neural networks and tree search,” *Nature*, vol. 529, pp. 484–503, 2016.

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the Game of Go without Human Knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm,” *arXiv preprint arXiv:1712.01815*, 2017.

[15] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model,” *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[16] J. R. Quiñones and J. Fernández Leiva, Antonio, “Automated video game parameter tuning with XVGDL+,” *Journal of Universal Computer Science*, vol. 28, no. 12, pp. 1282–1311, 2022.

[17] R. D. Gaina, R. Volkovas, C. G. Daz, and R. Davidson, “Automatic game tuning for strategic diversity,” in *2017 9th Computer Science and Electronic Engineering (CEEC)*, 2017, pp. 195–200.

[18] L. Kocsis, C. Szepesvári, and M. H. M. Winands, "RSPSA: Enhanced Parameter Optimization in Games," in *Advances in Computer Games*, H. J. van den Herik, S.-C. Hsu, T.-s. Hsu, and H. H. L. M. J. Donkers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 39–56.

[19] K. Chang, B. Aytemiz, and A. M. Smith, "Reveal-more: Amplifying human effort in quality assurance testing using automated exploration," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

[20] C. Dragert, J. Kienzle, and C. Verbrugge, "Statechart-based AI in practice," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 8, no. 1, pp. 136–141, Jun. 2021.

[21] H. Warpefelt and B. Strååt, "Breaking immersion by creating social unbelievabilty," in *Proceedings of AISB 2013 Convention. Social Coordination: Principles, Artefacts and Theories (SOCIAL. PATH)*, 2013, pp. 92–100.

[22] Wikipedia contributors, "List of video game genres — Wikipedia, the free encyclopedia," 2024, Accessed on 2024-04-13. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_video_game_genres&oldid=1211368541

[23] Ubisoft, "Rainbow Six Siege," Accessed on 2024-04-08. [Online]. Available: https://www.ubisoft.com/en-gb/game/rainbow-six/siege

[24] Irrational Games, "SWAT 4," Accessed on 2024-04-08. [Online]. Available: https://swat-4.fandom.com/wiki/SWAT_4

[25] VOID Interactive, "Ready or Not," Accessed on 2024-04-08. [Online]. Available: https://voidinteractive.net/

[26] Blizzard Entertainment, "Starcraft," Accessed on 2024-04-08. [Online]. Available: https://starcraft.blizzard.com/en-us/

[27] Sid Meier, "Civilization," Accessed on 2024-04-08. [Online]. Available: https://civilization.2k.com/civ/

[28] Subset Games, "Into the Breach," Accessed on 2024-04-08. [Online]. Available: https://subsetgames.com/itb.html

[29] Colossal Order, "Cities Skylines," Accessed on 2024-04-08. [Online]. Available: https://www.paradoxinteractive.com/games/cities-skylines/about

[30] Asobo Studio, "Flight Simulator," Accessed on 2024-04-08. [Online]. Available: https://www.flightsimulator.com/

[31] Eric ”ConcernedApe” Barone, “Stardew Valley,” Accessed on 2024-04-08. [Online]. Available: https://www.stardewvalley.net/

[32] Ludeon Studios, “Rimworld,” Accessed on 2024-04-08. [Online]. Available: https://rimworldgame.com/

[33] 11 bit studios, “Frostpunk,” Accessed on 2024-04-08. [Online]. Available: https://www.frostpunkgame.com/

[34] Creative Assembly, “Total War,” Accessed on 2024-04-08. [Online]. Available: https://www.totalwar.com/

[35] Capcom, “Street Fighter,” Accessed on 2024-04-08. [Online]. Available: https://www.streetfighter.com/

[36] Dan Fornace, “Rivals of Aether,” Accessed on 2024-04-08. [Online]. Available: https://rivalsofaether.com/

[37] Bandai Namco Entertainment, “Tekken,” Accessed on 2024-04-08. [Online]. Available: https://tekken.com/

[38] Blizzard Entertainment, “World of Warcraft,” Accessed on 2024-04-08. [Online]. Available: https://worldofwarcraft.blizzard.com/en-us/

[39] Pearl Abyss, “Black Desert Online,” Accessed on 2024-04-08. [Online]. Available: https://www.naeu.playblackdesert.com/

[40] Crema, “Temtem,” Accessed on 2024-04-08. [Online]. Available: https://crema.gg/games/temtem/

[41] Square Enix, “Final Fantasy XV,” Accessed on 2024-04-08. [Online]. Available: https://finalfantasyxv.square-enix-games.com/

[42] CD Projekt RED, “The Witcher,” Accessed on 2024-04-08. [Online]. Available: https://www.thewitcher.com/be/en/

[43] Toby Fox, “Undertale,” Accessed on 2024-04-08. [Online]. Available: https://undertale.com/

[44] Endnight Games, “The Forest,” Accessed on 2024-04-08. [Online]. Available: https://endnightgames.com/games/the-forest

[45] Studio Wildcard, “ARK: Survival Evolved,” Accessed on 2024-04-08. [Online]. Available: https://playark.com/ark-survival-evolved/

[46] Iron Gate Studio, “Valheim,” Accessed on 2024-04-08. [Online]. Available: https://www.valheimgame.com/

[47] Valve, "Portal," Accessed on 2024-04-08. [Online]. Available: https://developer.valvesoftware.com/wiki/Portal

[48] Playdead, "LIMBO," Accessed on 2024-04-08. [Online]. Available: https://playdead.com/games/limbo/

[49] Croteam, "The Talos Principle," Accessed on 2024-04-08. [Online]. Available: https://www.thetalosprinciple.com/

[50] Nintendo, "Super Mario Odyssey," Accessed on 2024-04-08. [Online]. Available: https://supermario.nintendo.com/

[51] Naughty Dog, "Crash Bandicoot," Accessed on 2024-04-08. [Online]. Available: https://www.crashbandicoot.com/

[52] Maddy Makes Games, "Celeste," Accessed on 2024-04-08. [Online]. Available: https://www.celestegame.com/

[53] S. A. Weil, T. S. Hussain, T. T. Brunyé, F. Diedrich, E. Entin, W. Ferguson, J. Sidman, L. Spahr, J. MacMillan, and B. Roberts, "Assessing the potential of massive multi-player games to be tools for military training," in *Proceedings of the interservice/industry training, simulation, and education conference (I/ITSEC)*, 2005.

[54] J. Linietsky and A. Manzur, "Godot Engine," Accessed on 2024-04-08. [Online]. Available: https://godotengine.org/

[55] Epic Games, "Unreal Engine," Accessed on 2024-04-08. [Online]. Available: https://www.unrealengine.com/en-US

[56] Unity Technologies, "Unity," Accessed on 2024-04-08. [Online]. Available: https://unity.com/

[57] Scirra, "Construct 3," Accessed on 2024-04-08. [Online]. Available: https://www.construct.net/en

[58] ASCII, Enterbrain, "RPG Maker," Accessed on 2024-04-08. [Online]. Available: https://www.rpgmakerweb.com/

[59] YoYo Games, "GameMaker," Accessed on 2024-04-08. [Online]. Available: https://gamemaker.io/en

[60] S. Sanshiro, "GODOT: The Open Source Engine Behind The Interactive Adventures of Dog Mendonça & Pizza Boy," October 31 2014, Accessed on 2024-04-03. [Online]. Available: https://www.gamingonlinux.com/articles/

godot-the-open-source-engine-behind-the-interactive-adventures-of-dog-mendon%C3%A7a-pizza-boy.4520/

[61] J. Linietsky, A. Manzur, and the Godot community, “Godot docs,” Accessed on 2024-04-08. [Online]. Available: https://docs.godotengine.org/en/stable/index.html

[62] A. Wilkes, “Signals in godot,” Accessed on 2024-04-08. [Online]. Available: https://gdscript.com/solutions/signals-godot/

[63] G. Docs, “Using Signals,” November 10 2023, Accessed on 2024-04-17. [Online]. Available: https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

[64] X. Cui and H. Shi, “A*-based pathfinding in modern computer games,” *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.

[65] K.-H. C. Wang, A. Botea *et al.*, “Fast and Memory-Efficient Multi-Agent Pathfinding.” in *ICAPS*, vol. 8, 2008, pp. 380–387.

[66] J.-D. historianX Perry, “Incubation: Exploration With a Plan,” June 18 2020, Accessed on 2024-05-10. [Online]. Available: https://www.riotgames.com/en/r-and-d-office/incubation-exploration-with-a-plan

[67] K. Railey, “Top 5 Mistakes Made by Indie Game Developers,” October 29 2021, Accessed on 2024-05-10. [Online]. Available: https://flowlab.io/lab/blog/indie-dev-mistakes

[68] W. Leblanc, “Video Games Stuck In Development Hell: Part 1,” November 1 2021, Accessed on 2024-05-10. [Online]. Available: https://www.gameinformer.com/2021/11/01/video-games-stuck-in-development-hell-part-1

[69] A. H. G. R. Kan and G. T. Timmer, “Stochastic global optimization methods part II: Multi level methods,” *Mathematical Programming*, vol. 39, pp. 57–78, 1987.

[70] M. J. D. Powell, “A direct search optimization method that models the objective and constraint functions by linear interpolation,” in *Advances in Optimization and Numerical Analysis*, ser. Mathematics and Its Applications, S. Gomez and J.-P. Hennart, Eds. Springer, 1994, vol. 275, pp. 51–67.

[71] Z. Zhang, "PRIMA: Reference Implementation for Powell's Methods with Modernization and Amelioration," 2023, Accessed 2024-05-12, DOI: 10.5281/zenodo.8052654. [Online]. Available: http://www.libprima.net

[72] Wikipedia contributors, "Field of view — Wikipedia, the free encyclopedia," 2024, Accessed 2024-05-29. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Field_of_view&oldid=1223601071

[73] R. Gunasekar and N. K. Chandramohan, "Development of image acquisition system to eleminate blind spot of a-pillar," *International Journal of Scientific Research & Management Studies*, vol. 8, pp. 145–151, 10 2018.

[74] S. M. LaValle, 2019, Accessed on 2024-05-29. [Online]. Available: https://msl.cs.uiuc.edu/vr/vrch5.pdf

[75] R. Lievrouw, "Applying Dynamic Game Difficulty Adjustment Using Deep Reinforcement Learning," Master's thesis, Ghent University, 2020.

[76] P. Studio Inc, "Phaser," Accessed on 2024-05-31. [Online]. Available: https://phaser.io/

[77] M. Groves and P. team, "Pixijs," Accessed on 2024-05-31. [Online]. Available: https://pixijs.com/

[78] J. Togelius and J. Schmidhuber, "An experiment in automatic game design," *2008 IEEE Symposium on Computational Intelligence and Games, CIG 2008*, pp. 111 – 118, January 2009.

[79] D. Silver, "Cooperative Pathfinding," *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 1, no. 1, pp. 117–122, Sep. 2021.

[80] N. de Barmon Vianney, "Coordination to Reduce the Impact of Friendly Fire," Académie Militaire de Saint-Cyr Coëtquidan, Tech. Rep., 2018.

[81] Department of the Army, "Army Training Publication (ATP) 3-21.8: Infantry Platoon and Squad," Accessed on 2024-06-01. [Online]. Available: https://www.moore.army.mil/Infantry/DoctrineSupplement/ATP3-21.8/

[82] lex92, "Review for Ready or Not," Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/id/lex92/recommended/1144200/

[83] SatKaz, "Review for Ready or Not," Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/id/satkaz/recommended/1144200/

[84] Prussian Wolf, “Review for Ready or Not,” Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/profiles/76561198078121853/recommended/1144200/

[85] AxeWould, “SWAT 4 user reviews - Metacritic,” Accessed on 2024-04-06. [Online]. Available: https://www.metacritic.com/game/swat-4/user-reviews/?platform=pc

[86] Welkore, “SWAT 4 user reviews - Metacritic,” Accessed on 2024-04-06. [Online]. Available: https://www.metacritic.com/game/swat-4/user-reviews/?platform=pc

[87] AlexFili, “SWAT 4 user reviews - Metacritic,” Accessed on 2024-04-06. [Online]. Available: https://www.metacritic.com/game/swat-4/user-reviews/?platform=pc

[88] DukeoftheAges, “Review for Spec Ops: The Line,” Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/id/DukeoftheAges/recommended/50300/

[89] ZarquonReturns, “Review for Spec Ops: The Line,” Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/profiles/76561198002009022/recommended/50300/

[90] GrampleGust, “Review for Spec Ops: The Line,” Accessed on 2024-04-06. [Online]. Available: https://steamcommunity.com/id/GrampleGust/recommended/50300/

Appendices

APPENDIX A

Additional figures

A.1 Game State Output

```
        {
        "allies": {
            "Ally": {
                "aim_direction": null,
                "ammo": 25,
                "goal_position": "(1515, 209)",
                "health": 100,
                "id": "Ally",
                "initial_locations": [
                "(1789, 523)"
                ],
                "path": [
                    "(1344, 224)",
                    "(1515, 209)"
                ],
                "position": "(1375.841, 224.0002)",
                "previous_state": 0,
                "reload_count": 0,
                "state": 2,
                "target": null},
            "Ally2": {
                "aim_direction": "(1692.697, 80.07452)",
                "ammo": 2,
                "goal_position": "(1515, 209)",
                "health": 100,
                "id": "Ally2",
                "initial_locations": [
                "(1789, 523)"
                ],
                "path": [
                    "(1408, 224)",
                    "(1472, 224)",
                    "(1515, 209)"
                ],
                "position": "(1416.16, 224)",
                "previous_state": 2,
                "reload_count": 0,
                "state": 1,
                "target": "Enemy3"}
        },
        "bullets": {
            "0": {
                "DIR": "(0.858848, 0.51223)",
                "POS": "(1983.232, 611.1429)",
                "SPEED": 10,
                "TEAM": 0
            }
        },
        "damage_done": {
            "allies": 260,
            "enemies": 0
        },
        "enemies": {
            "Enemy": {
                "aim_direction": null,
                "ammo": 10,
                "goal_position": "(1624.6, 387.2739)",
                "health": 0,
                "id": "Enemy",
                "initial_locations": [],
                "path": [
                    "(1624.6, 387.2739)"
                    ],
                "position": "(1626.91, 383.4973)",
                "previous_state": -1,
                "reload_count": 0,
                "state": 0,
                "target": null},
            "Enemy2": {
                "aim_direction": null,
                "ammo": 10,
                "goal_position": "(1648.884, 601.8615)",
                "health": 40,
                "id": "Enemy2",
                "initial_locations": [],
                "path": [
                    "(1632, 576)",
                    "(1648.884, 601.8615)"
                ],
                "position": "(1641.406, 588.5283)",
                "previous_state": -1,
                "reload_count": 0,
                "state": 0,
                "target": null}
        },
        "team_damage": {
            "allies": 0,
            "enemies": 0
        },
        "timer": 15.5498733333328
        }
```

APPENDIX B

Additional Tables

B.1 Combat Shooter Games - User Reviews

game	review snippet	user
Ready or Not	A great game held back by terrible **AI** [...].	lex92 [82]
	While SWAT **AI** is improved, they still have massive problems with reacting to gunfire [...].	satkaz [83]
	The **AI** is just so far removed from natural behavior that it completely ruins all immersion.	Prussian Wolf [84]
SWAT 4	Ill be honest [...] the intelligence of your partners is simply below par [...] you didnt pass the mission through your own efforts, but because the **AI** of your partners worked well and wasnt stupid [...].	AxeWould [85]
	Talking about **AI**, this is the dumbest **AI** I ever seen.	Welkore [86]
	The **AI** isnt perfect but often does a good job. Theyll often ask you to move out of the way which can be annoying [...].	AlexFili [87]
Spec Ops: The Line	[...] the friendly **AI** is dumb, so you will do all the work.	DukeoftheAges [88]
	Combat is jannnnkkkkyyyyyy as heck. [...] your **AI** companions stand around not killing things.	ZarquonReturns [89]
	Worst of all: Clumsy **AI** companions which run into full groups of enemies [...].	GrampleGust [90]

Table B.1: Table of negative user reviews mentioning AI for *Ready or Not*, *SWAT 4* and *Spec Ops: The Line*.