



MASTERARBEIT | MASTER'S THESIS

Titel | Title

Vienna Game AI Library

verfasst von | submitted by
Lavinia-Elena Lehaci

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of
Master of Science (MSc)

Wien | Vienna, 2024

Studienkennzahl lt. Studienblatt | Degree
programme code as it appears on the
student record sheet:

UA 066 921

Studienrichtung lt. Studienblatt | Degree
programme as it appears on the student
record sheet:

Masterstudium Informatik

Betreut von | Supervisor:

Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Univ.-Prof. Dipl.-Ing. Dr. Helmut Hlavacs, for his constant guidance and invaluable feedback throughout the course of my thesis.

I want to also thank my boyfriend, Tsvetelin, as his encouragement and never-ending support helped me navigate through the most challenging parts of this journey.

Lastly, I want to thank my parents and my sister, Irina, for their unconditional love and belief in me which motivated me to do my best and keep my morale high throughout this process.

Abstract

Real-time strategy games rely on artificial intelligence to provide immersive experiences, along with challenging and intelligent opponents for the player. The techniques that enable such aspects require significant effort if developed from scratch, thus it may be useful for game developers to have them integrated into a single framework. This thesis proposes a collection of features written in a C++ single-header file under the name of Vienna Game AI Library. It encompasses pathfinding using the A* algorithm on navigation meshes with geometric preprocessing and multithreading, decision-making processes such as decision trees and state machines, and steering behaviours for both individual and group units. Each individual feature is showcased in a separate demo that serves as an example for potential users of the library. This paper provides a detailed examination of the library from a development point of view and evaluates it on the basis of its applicability, performance, and usability. The findings of this paper demonstrate that the Vienna Game AI Library is developer friendly and offers efficient features that are sufficient to create a 2D real-time strategy game from an AI perspective.

Keywords

Artificial Intelligence, Video Games, Library, Real-Time Strategy, Algorithms, Non-Player Character, C++

Kurzfassung

Echtzeit-Strategiespiele stützen sich auf künstliche Intelligenz, um dem Spieler ein fesselndes Erlebnis mit herausfordernden und intelligenten Gegnern zu bieten. Die Techniken, die solche Aspekte ermöglichen, erfordern einen beträchtlichen Aufwand, wenn sie von Grund auf neu entwickelt werden, daher kann es für Spieleentwickler nützlich sein, sie in einem einzigen Framework integriert zu haben. Diese Arbeit schlägt eine Sammlung von Funktionen vor, die in einer C++ Single-Header-Datei unter dem Namen Vienna Game AI Library geschrieben wurden. Sie umfasst Pathfinding unter Verwendung des A*-Algorithmus auf Navigationsnetzen mit geometrischer Vorverarbeitung und Multithreading, Entscheidungsprozesse wie Decision Trees und Finite State Machines sowie Steuerungsverhalten sowohl für einzelne als auch für Gruppeneinheiten. Jede einzelne Funktion wird in einer separaten Demo vorgestellt, die als Beispiel für potenzielle Nutzer der Bibliothek dient. In diesem Beitrag wird die Bibliothek unter Entwicklungsgesichtspunkten eingehend untersucht und anhand ihrer Anwendbarkeit, Leistungsfähigkeit und Benutzerfreundlichkeit bewertet. Die Ergebnisse dieses Papers zeigen, dass die Vienna Game AI Library entwicklerfreundlich ist und effiziente Funktionen bietet, die für die Entwicklung eines 2D-Echtzeitstrategiespiels aus KI-Sicht ausreichend sind.

Contents

Acknowledgements	i
Abstract	iii
Kurzfassung	v
List of Tables	ix
List of Figures	xi
List of Algorithms	xiii
Listings	xv
1. Introduction	1
1.1. Motivation	2
1.2. Research Questions	2
1.3. Synopsis	2
2. Related Work	4
2.1. Literature Review	4
2.2. Existing Software	8
3. Background	11
3.1. Game Concepts	11
3.2. Game AI Concepts	13
4. The Vienna Game AI Library	18
4.1. Design	18
4.2. Implementation	24
5. Evaluation and Discussion	46
5.1. Applicability Evaluation	46
5.2. Performance Evaluation	48
5.3. Usability Evaluation	58
5.4. Discussion	62
6. Conclusion and Future Work	67

Contents

Bibliography	69
A. Appendix	79
A.1. Survey Questions	79
A.2. Survey Results	87
A.3. Wilcoxon Rank-Sum Test Results	103

List of Tables

5.1. Standard vs. Improved Pathfinding: Execution Times	50
5.2. “Seek” and “Flee” Behaviours: Execution Times	55
5.3. “Pursue” and “Evade” Behaviours: Execution Times	55
5.4. “Arrive” Behaviour: Execution Times	55
5.5. “Wander” Behaviour: Execution Times	56
5.6. “Face” Behaviour: Execution Times	56
5.7. The Flocking Algorithm: Execution Times	57
5.8. Wilcoxon Rank-Sum Test: Results	62

List of Figures

4.1. UML Diagram for Pathfinding	21
4.2. UML Diagram for Decision Trees	22
4.3. UML Diagram for State Machines	22
4.4. UML Diagram for Steering Behaviours	23
4.5. Geometric Preprocessing vs. Standard A* Algorithm	29
4.6. Demo for Decision Trees	33
4.7. Demo for State Machines	35
4.8. Wander Behaviour Diagram	40
4.9. Demo for Steering Behaviours	42
4.10. Demo for The Flocking Algorithm	45
5.1. Decision Trees Demo: Burglar's Decision Tree	52

List of Algorithms

1. The A* Algorithm: Pseudocode	27
---	----

Listings

4.1. Pathfinding with VGAIL: Code Snippet	31
4.2. Decision Trees with VGAIL: Code Snippet	32
4.3. State Machines with VGAIL: Code Snippet	34
4.4. Steering Behaviours with VGAIL: Code Snippet	36
4.5. “Seek” Behaviour: Code Snippet	37
4.6. “Arrive” Behaviour: Code Snippet	38
4.7. “Pursue” Behaviour: Code Snippet	38
4.8. “Face” Behaviour: Code Snippet	39
4.9. “Face” Behaviour with VGAIL: Code Snippet	40
4.10. “Wander” Behaviour: Code Snippet	41
4.11. Separation Behaviour: Code Snippet	43
4.12. Align Behaviour: Code Snippet	43
4.13. Cohesion Behaviour: Code Snippet	44
4.14. The Flocking Algorithm with VGAIL: Code Snippet	44
5.1. Wilcoxon Rank-Sum Test: Code Snippet	61

1. Introduction

Since the early days of video games, artificial intelligence (AI) has been a significant part of game development as it shapes player interactions and enhances game experiences. Whether it is used to generate textures or meshes, to simulate realistic character behaviour or for optimization purposes, AI plays a crucial role in the gaming industry. In [1], Karpouzis and Tsatiris explore the use cases of AI in games and demonstrate that it fosters adaptive player experiences and is relevant for maintaining player enjoyment. Many games have expanded on the idea of intelligent AI, starting from the classics *Space Invaders* [Taito Corporation, 1978], *Pac-Man* [Namco Limited, 1980] or *Sid Meier's Civilization* [MicroProse, 1991], to the more modern ones such as *Alien: Isolation* [Creative Assembly, 2014], *Red Dead Redemption 2* [Rockstar Games, 2018] or *Age of Empires IV* [Relic Entertainment, 2021].

Artificial intelligence continues to be used in game development, as it is present in action, adventure, strategy, sports, and simulation video games. In particular, real-time strategy (RTS) games rely on AI in order to provide challenging and intelligent non-player characters (NPCs), which is done by ensuring realistic and natural character movement and by simulating complex decision-making processes. In this paper, the terms “non-player character” and “AI agent” are used interchangeably, and refer to any AI-controlled character within the game environment. Jiang Jie et al. [2] mention two types of AI, namely strong and weak. Strong AI refers to machines with self-consciousness, while weak AI is attributed to machines that just look intelligent, but have no consciousness. Currently, only the latter is possible, and is represented in strategy games through NPCs as these only imitate human behaviour, but have no thinking on their own.

While each RTS game is different, there is a set of AI features that this genre benefits from, such as resource management, pathfinding, procedural content generation, and decision-making processes. A collection of such features would empower game developers with AI algorithms that are customizable enough to fit their games' requirements. Currently, many algorithms are implemented individually or are part of custom projects, but there is no public library written in the C++ programming language that encompasses a standard set of features required to introduce and manage AI agents in RTS games.

The Vienna Game AI Library (VGAIL) is a proposal to fill this gap and provides a comprehensive and easy-to-use collection of algorithms tailored specifically for the AI development in real-time strategy games. From pathfinding and steering algorithms, to state machines and decision trees, this library should encompass the fundamental features a game developer might need in creating a 2D RTS game. In order to show how to use the library, each provided algorithm will be showcased in a respective demo implemented with the help of the Raylib [3] library. Each demonstration represents a use case in which the Vienna Game AI Library could be used to solve a problem.

1. Introduction

1.1. Motivation

In game development, C++ is one of the most used programming languages because it offers control over hardware, such as graphics processing units (GPUs), it is cross-platform and it is very efficient in speed and memory execution, allowing for many optimization possibilities. Therefore, a library written in this programming language would not only offer the aforementioned advantages, but would also be easy to integrate into C++ projects.

One core motivation behind the creation of this library is the lack of public collections of algorithms that could be used when creating a 2D real-time strategy game in C++. The next chapter gives insight into existent frameworks that provide similar features as the Vienna Game AI Library, with several being implemented in other programming languages, which makes them difficult to integrate into C++ projects. Others, discussed in the same section, appear to be either incomplete or no longer maintained.

Another factor contributing to the development of this library is the need of such a collection within the projects used in courses at the Faculty of Computer Science of the University of Vienna. Currently there are a few projects that deal with game physics, entity component systems, as well as a rendering engine, all of which could be used together with an AI library to enhance the opportunities for students to develop games. The name of the library is derived from its association with this collection of projects.

1.2. Research Questions

To evaluate the goals of the Vienna Game AI Library, the following research questions are proposed:

- **RQ1:** Does the proposed library provide all features required to develop 2D real-time strategy games from an AI perspective?
- **RQ2:** Are the features provided by the proposed library easy to use?
- **RQ3:** Is the proposed library well documented and demonstrated?
- **RQ4:** Is the proposed library easy to integrate into C++ projects?

1.3. Synopsis

This section presents the structure of this paper by examining each chapter.

Chapter 1 offers a high-level introduction of the AI usage in game development, while also stating the problem addressed by this thesis. It covers the motivation behind the Vienna Game AI Library, as well as the research questions that are used in evaluating the success of this project. Additionally, it summarises the structure of this paper.

Chapter 2 covers the work conducted on the topic introduced by this thesis. It presents published papers on well-established AI domains that are relevant for the content of this project, as well as scientific papers that discuss AI algorithms used in game development.

Moreover, it provides an overview of public software that shares similarities with the proposed library.

Chapter 3 highlights the foundational knowledge that is necessary for understanding the goal and content of the library. It explains a series of fundamental game concepts that are essential in grasping the terminologies used throughout this paper. It also discusses various AI features and algorithms.

Chapter 4 presents the Vienna Game AI Library from different perspectives. First, it outlines the design of the library, covering its setup, content and architecture. Additionally, it explains the rationale behind choosing each feature that is offered by the library. Lastly, it dives into the implementation of each chosen AI feature, by providing extensive explanations and code snippets.

Chapter 5 illustrates three ways in which the library is evaluated in order to answer the research questions stated earlier in this chapter. It also investigates the results obtained from these tests and their relevance to the library's goals. Furthermore, it outlines potential improvements and optimizations that could enrich each individual feature, along with limitations that need to be considered.

Chapter 6 concludes the work presented in this thesis. It offers a final reflection on the Vienna Game AI Library by summarising the major points discussed throughout this paper, and by stating its plans for future work.

2. Related Work

AI algorithms and techniques have been implemented by developers for several years whether as stand-alone projects or as part of different collections. As outlined in the previous chapter, one of the motivations behind the Vienna Game AI Library is the lack of public libraries implemented in C++ that contain enough features to develop games of a certain genre, especially real-time strategy.

There are several frameworks similar to the Vienna Game AI Library, but they either are implemented in other programming languages, or lack algorithms that are crucial in developing 2D RTS games. An overview of such frameworks is given in this chapter, as well as a summary of the scientific papers that have been written on this topic.

2.1. Literature Review

In many game genres there are instances when non-player characters need to navigate through the game world from one location to another in the most efficient way. Sometimes, the path to reach the target location also involves various obstacles that the AI agent needs to avoid. To address this problem, many pathfinding algorithms have been developed and continue to be used to this day, such as Dijkstra's algorithm, Breadth-First Search, and Depth-First Search. However, since game engines and games are real-time applications and cannot afford having any disruptions at runtime, game developers have started to prefer the A* algorithm [4] and its variations instead. They are much more efficient, especially in scenarios involving large, complex, even dynamic environments.

Although the A* algorithm is one of the most popular search algorithms in the gaming industry, optimising its efficiency can be challenging. For instance, since the algorithm relies on an internal data structure to calculate the path, the larger and more complex this data structure is, the more challenging it is to ensure smooth and efficient performance.

Cui et al. [5] give an overview of two well-known A*-based algorithms that address this issue, namely Hierarchical Pathfinding A* (HPA*) and Navigation Mesh (NavMesh). The former aims to decrease complexity by dividing the surface in a hierarchical manner, while the latter uses a set of convex polygons that describe the surface. The same authors also mention challenges in regards to the memory usage, stating that the algorithm requires much space to store the processed search information.

One other popular A* variant that is used to reduce memory usage is Iterative Deepening A* (IDA*), which computes the path in steps by gradually increasing a cost limit. However, Primanita et al. [6] found that IDA* is generally better than the classic A* algorithm if the game map has no obstacles, making it less suitable for more complex maps.

Dynamic game environments represent a challenge in strategy games as well, since this genre typically implies large-sized game worlds with obstacles that can change over time. To address this challenge, some propose pathfinding planning algorithms that are based on the A* algorithm, such as Lifelong Planning A* (LPA*) [7], Anytime Dynamic A* (AD*) [8] or Real-Time Adaptive A* (RTAA*) [9]. Each optimization algorithm has its own advantages and disadvantages, and its suitability depends heavily on the game’s requirements and priorities. However, a good performance does not rely only on choosing specific algorithms. Bleiweiss [10], for instance, offers an efficient implementation of global pathfinding on the GPU by diving into data parallelism.

Non-player character movement does not imply only pathfinding. In game development, depending on the game genre, it is also important to have the option to steer AI agents along paths while possibly avoiding other agents or obstacles. Strategy games especially require such a feature, whether it is for individual or group units, in order to offer precise unit movement control.

When it comes to individual movement, a core principle in game AI development is represented by Craig Reynolds’s steering behaviours [11, 12]. These techniques are critical in creating realistic AI agents, and they can be used either separately, to achieve a specific behaviour, or together to create a more convincing movement. The most fundamental individual behaviours include “seek”, “flee”, “pursue”, “evade”, “arrive”, “wander”, and “obstacle avoidance”. They deal with directing the AI agent towards or away from a target, moving randomly across the game world or around obstacles, and approaching a target smoothly. Depending on the game requirements, some are more useful than others, and sometimes game developers combine them to achieve certain goals. The flocking algorithm [13, 14], one of the most popular combinations of steering behaviours, is built upon three key behaviours, namely “separation”, “avoidance” and “cohesion”. These techniques help in simulating natural movement for a group of units, a key aspect of strategy games. In this genre, units are required to move together in coordinated groups while ensuring that they maintain an average velocity, are well-positioned inside the group and do not overlap. Fathy et. al [15] investigate this behaviour in RTS games and propose an efficient technique to improve unit movement and decrease unit lose.

The ability to provide intelligent NPCs that can take decisions in real-time is also a significant requirement in several game genres. In strategy games, AI agents need to create, gather, and prioritise resources, build units, decide whether to fight or run, and even make strategies based on their opponents’ actions. There are many decision-making systems that developers use to implement such features, such as state machines, decision trees, fuzzy logic, goal-oriented behaviour, and many more. Millington and Funge [16] point out that these systems act in a similar manner: the AI agent possesses internal and external knowledge, upon which it can make a decision on what action it should carry out. External knowledge refers to the game environment or other characters, while internal knowledge is information about the agent’s internal state, such as health, goals, or combat skills.

Among the simplest decision-making systems are decision trees [17], which are similar to the tree data type as they are made of nodes, each of which can have its child nodes.

2. Related Work

While they are easy to implement and understand [16], there are cases when AI agents will need to choose and carry out an action from a limited given set, and in this case decision trees are not the most suitable solution. For this kind of behaviour, a fundamental and widely used system is typically chosen, namely finite state machines (FSMs) [18]. This simple, yet effective technique allows characters to carry out actions based on the state they are in, and to switch states when certain conditions are met. While they are a helpful tool in controlling the decision-making flow, they might impact performance in games if the number of states is too high. In order to prevent such a risk and state redundancy, as well as to express more types of behaviours, game developers sometimes resort to hierarchical state machines (HSMs) [19, 20], which are seen as an optimised version of FSMs [21].

Behaviour trees [22] resemble HSMs, with the difference that instead of states, they use tasks [16]. They are a popular alternative to FSMs and its variations, as they are more suitable in dynamic environments, although they imply certain limitations as well. Depending on the game genre and requirements, some decision-making systems are more appropriate than others, regardless of their popularity throughout the years. In the real-time strategy subgenre, game developers and researchers have also started to explore the Monte Carlo tree search (MCTS) algorithm to improve performance [23], even though it was previously used mostly in turn-based games.

The techniques covered in this section not only can be used individually to achieve specific goals, they can also be combined to enhance the realism and efficiency of AI behaviours. Steering behaviours, for instance, are often used together with pathfinding algorithms to create smooth movement in complex environments. Tomlinson [24] and Kapadia and Badler [25] investigate such combinations, while also outlining possible limitations or improvements. Danielsiek et. al [26] demonstrate that the flocking algorithm combined with influence map-based pathfinding algorithms improves unit performance in every tested game situation. Additionally, in [27], Naveed et. al discuss how the MCTS decision-making algorithm can be used to solve pathfinding problems more effectively than the classic search algorithms.

Although no papers discussing game AI libraries were found, several provide valuable insights into AI frameworks and algorithms. Some focus on NPC management, others offer simulation environments where AI algorithms can be developed and tested, all being highly relevant in game AI development.

One of these papers is “Development of Non-Player Character for 3D Kart Racing Game Using Decision Tree” [28], where Mas’udi et al. make use of pathfinding and decision trees to improve the behaviour and intelligence of non-player characters in a 3D Kart racing game. They use the waypoint system and raycasting to facilitate pathfinding and obstacle detection. A series of tests were conducted, such as white and black box testing, or in regards to frames per second (FPS) and lap time performance, that demonstrated the effectiveness of the chosen algorithms, as well as an increased fun factor that NPCs typically ensure. While the topic of the paper is not considered a library, it comprises a set of useful AI algorithms which, combined, can cover the NPC management in any racing game.

Miyake et al. [29] discuss a complex decision-making system, called AI Graph, that was applied to non-player characters in *Final Fantasy XV* [Square Enix, 2016]. A combination between behaviour trees and state machines created an advanced AI system that controls how characters make decisions and conduct actions based on their environment, while focusing on ensuring scalability and reducing code redundancy. Although it is not considered a library and it is not available for public use, the AI Graph system can serve as inspiration when creating scalable decision-making systems.

In “Application of behaviour tree in AI design of MOBA games” [30], Lin et al. explore the usage of behaviour trees when designing AI systems for multiplayer online battle arena (MOBA) games. They also discuss state machines and provide insights into how they differ from behaviour trees, highlighting the benefits the latter bring. The paper does not address implementation aspects, as it explores the tree only from a structural perspective. However, it offers valuable research and suggestions on the creation and usage of behaviour trees for non-player characters. Therefore, this paper is highly relevant in game development, especially for the MOBA genre.

In [31], Balapriya and Srinivasan address the challenge of designing efficient patrolling AI-driven agents by introducing two algorithms. One is a custom pathfinding algorithm based on two sampling-based AI algorithms, namely Probabilistic Roadmap (PRM) and Rapidly Exploring Random Trees (RRT). The other is an obstacle modelling algorithm that analyses dynamic elements in the game environment. Their research found these algorithms to perform well, although they mention high consumption of resources.

Shaout et al. [32] remake the famous *Pac-Man* game to showcase a real-time AI system based on fuzzy logic. They thoroughly discuss design and implementation aspects, as well as the results that prove the efficacy of the system. Not only the AI agents were successfully depicted as having intelligent behaviour, they also adapted easily to the player’s skill level.

Fronek et al. [33] present a machine learning approach to procedurally generate behaviour trees in order to manage decision-making processes and actions of non-player characters in a *Capture the Flag* game. The authors use algorithms and procedural techniques to generate such systems that can adapt to various game requirements. The game chosen to test this approach has a turn-based approach, consisting of commanding units and strategic decision-making processes typical of the strategy genre.

Another paper that targets the automatic generation of behaviour trees for non-player characters is “Shaping AI Behaviour: A Q-Learning Driven Approach to Automatic Behaviour Tree Creation” [34]. Similarly to [33], the authors developed a *Capture the Flag* game to test the proposed algorithm. They use Q-learning, a reinforcement learning technique, to train the AI agents in learning to make optimal decisions based on the expected rewards of their actions. The proposed algorithm aims to provide an option to efficiently develop intelligent NPC behaviour that can adapt to dynamic environments.

Although it does not address AI algorithms, PALAIS [35] is a virtual 3D simulation environment that is designed specifically for AI research in game development. It offers developers and researchers the ability to create and modify scenarios such that they can develop and evaluate AI algorithms in a dynamic and interactive 3D environment.

2. Related Work

It is a comprehensive tool tailored specifically for the needs of a game developer as it provides support for both interpreted and native code. The platform is also connected to an external pathfinding module that is based on the A* algorithm, highlighting the potential of this platform.

Similarly, “A Tactical and Strategic AI Interface for Real-Time Strategy Games” [36] is not centred around algorithms, but it provides valuable insights and enhances the real-time strategy subgenre through the interface it proposes. It offers an AI framework that addresses key aspects of decision-making, such as tactical unit control, resource allocation, base-building, and long-term planning. It also mentions challenges that are relevant when developing tactical and strategic AI systems, namely terrain analysis and multi-agent coordination of plans. The paper is relevant in game AI development as it covers relevant information about the depth and complexity of the RTS subgenre. However, the current status of the proposed framework is unclear, as the paper stated it to be under development in 2004.

There are also papers that discuss machine learning techniques to develop game AI features, such as [37] on generating intelligent agent behaviour using reinforcement learning and [38] on combining neural networks to control non-player characters. OpenAI Gym [39] is also a notable mention, as it is a comprehensive toolkit used in reinforcement learning research. Given that the plans and goals for the Vienna Game AI Library do not currently cover such features, further papers related to them will be omitted.

2.2. Existing Software

Upon researching existing systems that share similarities with the Vienna Game AI Library, various online projects were discovered, which were either published for commercial use or uploaded as individual, open-source projects to serve as inspirations for other developers. This section presents the frameworks that come close to, or even exceed, the main goals of the Vienna Game AI Library.

One of the most extensive AI toolkits seems to be Yuka [40], a JavaScript library that provides a comprehensive set of AI algorithms designed for game development. It does not seem to target a specific game genre for development, but it offers various features that can be used in different types of games, making it a suitable toolkit for the strategy genre. It covers areas such as fuzzy logic, navigation, maths, and steering behaviours, while also showcasing their usability through a series of informative and sometimes interactive examples. Due to this, the library served as inspiration for the Vienna Game AI Library in the early planning stages. While this library could be one of the most comprehensible game AI libraries discovered, due to the lack of compatibility between JavaScript and C++ and the fact that it is developed for web-based video games, it is not a suitable option for developing native, cross-platform or stand-alone games in C++.

gdxAI [41] is also a remarkable collection of algorithms, very similar to the proposed library as it supports pathfinding, steering, both individual and group, as well as decision-making. It also offers infrastructure features, such as scheduling and message handling, making it an extensive and suitable toolkit for the development of the strategy genre.

However, it is entirely written in JAVA and it was initially developed to be used only with the libGDX[42] framework, although now it is possible to use it separately. Nonetheless, incorporating it into C++ project might be challenging, but it can still be used as further inspiration for future improvements or plans for this project.

Another example of a robust game AI framework is BrainAI [43], which appears to be no longer maintained since there are no changes made in recent years. This library provides a wide range of algorithms across four areas, with multiple features available in each. For instance, in the decision-making area, it includes techniques such as finite state machines, behaviour trees, goal-oriented action planning (GOAP) and utility-based AI. Similarly, the library provides several options for pathfinding and simulations, in addition to support for influence maps, and showcases dynamic examples of some of these features by using the Godot engine. While it is an extensive collection, it arguably lacks a feature that could be relevant in the development of the strategy genre, namely steering behaviours. Moreover, it is written in C#, which makes the process of integrating it into C++ projects cumbersome due to the differences in runtime environments.

Another C# library is Unity Movement AI [44] which is focused solely on AI steering techniques, offering various algorithms that can be used in any type of games to manage AI agent movement. It does not seem to be maintained anymore, but it contains extensive functionality, around 15 steering behaviours, as well as small scenarios to illustrate their usability. Although extensive in one area, it lacks support for others that are relevant in game development, and it cannot be used natively in a simple manner with C++ projects.

Although written in Python, Pygame [45] is a powerful and broad library that seems to cover any game genre, but not necessarily from an AI perspective. It primarily focuses on rendering, sound and user input, but provides a foundation for users to build AI techniques on, regardless of their platform or operating system. The library's event handling system allows users to manage game states and transitions, making it possible to implement state machines or behaviour trees. It also offers basic collision detection on which users can work on to implement collision, obstacle, and wall avoidance. While it is a useful toolkit suitable for many game genres, it does not include actual AI functionality and requires developers to create such features on their own.

A C++ framework that could be used in game AI development is behaviac [46]. However, it does not seem to be maintained anymore, as its last visible changes were made two years ago. It is specified that it can be used as a "rapid game prototype design tool", and its functionality is centred around decision-making systems. Therefore, it supports features such as behaviour trees, finite state machines, and hierarchical task networks, which do not cover all requirements needed in the development of the strategy genre.

Recast Navigation [47], also written in C++, is a navigation mesh package, tailored for agent movement. It contains multiple modules that support features such as navigation mesh generator, pathfinding, and agent movement, as well as collision avoidance and crowd simulation. It is a solid library, and arguably broad enough to be used in developing RTS games. The library resembles the features targeted by the Vienna Game AI Library, and therefore is a viable alternative. However, it lacks decision-making systems, essential when managing NPCs behaviour, making this thesis's proposed solution more comprehensive.

2. Related Work

Initially developed by Craig Reynolds, OpenSteer [48] is a C++ framework that handles steering behaviours for autonomous characters. It is cross-platform and it provides predefined and interactive demonstrations of its features. The library covers a wide set of steering behaviours, including flocking, obstacle avoidance, path following, and unaligned collision avoidance. Similarly to Recast Navigation [47], it is an extensive option that can be used to manage NPC movement, but it lacks pathfinding and decision-making systems, making it insufficient in developing real-time strategy games.

The built-in support for pathfinding, behaviour trees and steering behaviours from the Godot engine [49] is also a relevant alternative in game development. The algorithms provided by the engine are arguably enough to create a real-time strategy game, therefore it can be stated that it can fully cover at least one game genre. These features are part of the core engine and are written in C++, but users have the option to choose different supported languages for development, including GDScript, C# or VisualScript. While it is possible to integrate C++ code into Godot, these AI features are deeply incorporated into the engine's architecture, meaning that users have to build their own features anew if they do not wish to use the engine. For developers, such as the students at the University of Vienna who most likely have to use a different engine, these options are not the most favourable.

The Free Fuzzy Logic Library [50, 51] is an open-source class library and application programming interface (API) that is designed to be fast, thus suitable for game development. Its focus is on fuzzy logic, while also offering features such as multithread support, model loading, and Unicode support. It promises to save significant time whether it is used in an AI engine or for prototyping. While it is an appropriate option to be considered for the specified features, it can be stated that it does not provide enough functionality to cover the development of any game genre.

The libraries mentioned in this section are the most comprehensive ones that are relevant for the topic of this thesis. Although a few projects developed for game genres other than strategy were mentioned, it is believed that their content is relevant for this thesis, as they share similarities with the Vienna Game AI Library in one form or another. There are many other interesting projects, such as GameAI [52], which targets the turn-based strategy subgenre by offering the MiniMax and MCTS algorithms, or Unity Machine Learning Agents Toolkit [53] that focuses on training intelligent agents using reinforcement learning and imitation learning. However, their focus and content, regardless of the game genre they target, do not align with the goals of this project, hence they will not be presented in more details.

Having discussed existent research and software, it is anticipated that enough information has been provided to confirm the lack of public game AI libraries, written in C++, thus solidifying the core motivation behind the Vienna Game AI Library.

3. Background

Before diving into the technical part of this paper, it is important to understand all characteristics that are relevant for the content of the library. Therefore, this section will provide fundamental knowledge that will explain the terminologies mentioned throughout the paper. General game concepts will be reviewed, in addition to AI domains and algorithms.

3.1. Game Concepts

This section begins by exploring video games from a high-level perspective and by discussing their general characteristics, purpose, and the different genres they belong to. It is important to note that, throughout this paper, the terms “game” or “video game” refer to only computer games.

Video games are an interactive form of entertainment that people play on their personal devices (in this paper this refers to computers only) individually or with others, either cooperatively or competitively. They are proven to be motivational and socially interactive, while also having the potential to promote well-being by preventing and treating mental health problems [54]. Additionally, they are a means for people to unwind, express creativity and prevent boredom.

While each game differs in narratives and aesthetics, they are primarily distinguished by their genre. Most popular game genres include role-playing, action, shooter, strategy, simulation, and sports, each of which consisting of subgenres of their own. Role-playing games can be massively multiplayer online (*World of Warcraft* [Blizzard Entertainment, 2004]), or tactical (*Final Fantasy Tactics: The War of the Lions* [Square Enix, 2007]). Strategy games are available in real-time (*Age of Empires IV* [Relic Entertainment, 2021]), and turn-based (*Sid Meier’s Civilization VI* [Firaxis Games, 2016]) formats. Shooter games can be first-person (*Valorant* [Riot Games, 2020]), or third-person (*Fortnite* [Epic Games, 2017]). Simulation games include life simulation (*The Sims 4* [Maxis, 2014]), and sandbox (*Minecraft* [Mojang Studios, 2011]). These are just a few examples among the vast spectrum of game genres and subgenres that exist.

Game development is the process of creating games of such genres, and it comprises various areas, such as programming, testing, design, sound, art, and publishing. Its purpose differs from individual to individual, or from company to company. For players, games may offer entertainment and learning opportunities, as well as improve cognitive skills such as strategic thinking and hand-eye coordination. For developers, games represent means to refine their technical and problem-solving skills, while also encouraging creativity. From the gaming industry’s perspective, games provide jobs and revenue.

3. Background

A fundamental aspect of game development is the game engine. According to Gregory [55], the term “game engine” refers to software that is expandable and can be used as the foundation for creating many different video games without involving major engine changes. It generally includes different tools and frameworks that support features such as rendering, physics, sound, scripting, networking, and many others. Building a game engine anew can be challenging due to the amount of features it needs to provide, which is why game developers often choose to use existing ones. There are a few game engines that have grown in popularity in game development and are used more often than others. One of them is Unity [56], which uses C# as the scripting language and is used to create video games, simulations and even virtual reality (VR) applications. It is a powerful and flexible cross-platform engine, used for both 2D and 3D game development. Another widely used cross-platform game engine is Unreal Engine [57], developed by Epic Games, which is known for the high-quality graphics it provides. It is designed to script with Blueprint Visual Scripting or natively with C++, and is mainly meant for 3D projects. Godot Engine [49] is also a popular alternative within game development as it provides several scripting languages such as C#, GDNative, and VisualScript. It primarily uses GDScript, its own built-in scripting language and, similarly to Unity, it provides support for both 2D and 3D game development.

One integral part of game engines is the rendering or graphics engine, which is a software component that is responsible for drawing 2D and 3D graphics on the screen. 2D graphics are flat images that are represented in two dimensions with their width being on the X-axis, and their height on the Y-axis. In game development they are usually represented by textures, often referred to as sprites, depicting characters, backgrounds, or objects. 3D graphics look more realistic as they have a third dimension, namely depth, added to them. These can be 3D models or visual elements, such as lights, shadows, and particles, that interact with the 3D models. Usually these graphics require complex calculations in order to be accurately displayed from the player’s perspective. In order to create and manage this type of visual elements, game engines often use graphics APIs, such as Vulkan [58], OpenGL [59] or Metal [60], that are in charge of sending all rendering data to the GPU for faster real-time processing. The Vienna Vulkan Engine [61], which is one of the Vienna projects mentioned in the introduction of this paper, serves as an example as it is a Vulkan-based rendering engine.

Another relevant component of game engines is the physics engine that is in responsible for simulating physical systems, such as gravity, fluid dynamics, collisions, and friction. It mimics the laws of physics in order to make elements within the game environment look and act natural, as they would in the real world. The physics engine uses mathematical operations to calculate various forces such as acceleration and velocity, which are necessary to create realistic movement. Additionally, such algorithms are essential in detecting and managing interactions between different objects, a process commonly known as collision detection. An example of such an engine is the Vienna Physics Engine [62], part of the Vienna projects, which provides features such as rigid body simulations, friction, and joint constraints. It is a single-header C++ library which can be used with any engine, especially with the Vienna Vulkan Engine, the rendering engine mentioned previously.

One of the goals behind the Vienna Game AI Library is ensuring that it integrates well with these two projects, such that they can collectively provide a comprehensive set of features for students to use in game development.

Two terms that are used in this paper with regards to games in general are “game state” and “game loop”. The game state is the status of the game at any given moment in time. It consists of any variables and details that are relevant for the progression of the game, such as characters’ positions within the game environment, or the player’s health level or inventory items. The game state can change over time as the player makes decisions and conducts actions. The term “game loop” refers to the cycle of updating the game state. During each iteration of the cycle, different game systems, for instance the game logic and physics simulations, process input, calculate and update their next steps such that these can be rendered on the screen [55]. These continuous iterations lead to smooth gameplay as they react to user input and update accordingly in real-time.

One last concept that needs to be clarified as it will be mentioned throughout this paper is multithreading [63, 64]. This technique is critical in game development as it allows complex computations to be carried out without blocking the main game loop. It works by using multiple threads at the same time, independently of each other, which leads to a parallel execution of tasks and therefore, to an improved performance. A thread is a small unit of execution which, in a program, works as a set of programmed instructions. While threads work independently of each other, they might share the same memory, thus introducing problems in synchronisation. Accessing the memory while other threads are writing to it causes data racing and results in inconsistent data. This problem is addressed by implementing synchronisation mechanisms such as locks and mutexes, or by dividing the data between the threads such that there is no shared memory. When developing rendering engines, the use of fences and semaphores are necessary for the synchronisation between the central processing unit (CPU) and the GPU.

3.2. Game AI Concepts

This section will delve into the fundamentals of AI in game development, covering basic terms, various algorithms and techniques, as well as several challenges and limitations.

Reynolds [11] states that AI focuses on “making computers able to perform the thinking tasks that humans and animals are capable of”. The field of artificial intelligence is broad and encompasses various concepts, each having its own objectives, benefits, as well as limitations. Game AI refers to AI specifically used for video games, whether that implies development, design, or testing, and aims to offer realistic and immersive game experiences. Its usage differs from area to area. In design, it is used to generate environments and scenarios, while in testing, AI can help detect bugs, as well as facilitate automated testing. AI is also suitable for ensuring immersive game experiences for the player, as it adjusts the game difficulty in response to the player’s actions and skill level.

In game development, AI is often used to simulate intelligent and challenging characters that can react and adapt to the game environment or to the actions of the player. The type of such characters typically depends on the game’s requirements, goals, or story.

3. Background

They can be either allies or enemies, in which case they would have some sort of contact with the player, but they can also be neutral characters, such as civilians or animals, which would react to the game environment without interacting with the player. Regardless of their type, AI agents have the role of creating an immersive and challenging game experience through their decision-making capabilities and interaction with the game world. To achieve this goal, game developers use various AI techniques in relation to decision-making, movement, and learning.

Decision-making AI deals with modelling NPC behaviour. It enables them to make decisions and conduct actions in a realistic manner based on various factors, such as their state, the environment or the actions of the player. There are many AI techniques that support such processes, each being suitable for different scenarios.

One system that stands out for its structured nature is the state machine, which is defined by a set of transitions and states. States are specific statuses of the state machine at any given time, and consist of a set of actions that the AI agent needs to perform. Transitions manage switching from one state to another, as they check if certain conditions are met or if specific game events occur. When the state machine is activated, the AI agent will be assigned an initial state, which could change throughout the game. The state of the agent at a given moment is referred to as the current state, and it is not possible for the agent to be assigned more than one state at the time. This approach is widely used in game development, as it is simple to understand and implement, as well as efficient in modelling NPC behaviour. Depending on the number of states that the machine has, it can be either finite or infinite. A finite state machine has a limited, predefined set of states, while an infinite machine consists of an unlimited number of states that could be created dynamically at runtime.

Two widely used systems are decision trees and behaviour trees, both resembling the tree data structure. Decision trees are an organised set of if-else statements, and depending on the number of children a node can have, they can be either binary or n -ary. The algorithm starts at the root node, evaluates the decision that is linked to it, then makes a choice based on the result. This leads to one of its child nodes where the process repeats, continuing in this manner along the tree until it reaches a node without children, also referred to as a leaf node. Decision trees are popular in game development due to their modularity and simplicity [16]. Behaviour trees are preferred for managing more complex decision-making processes. They have a hierarchical structure in which a node can be either the root, a control node, or an execution node. The latter can be in one of three different states when the algorithm is running: “success”, “failure”, and “running”. “Success” is used when the node achieved its goal, “failure” implies otherwise, while “running” means that the respective node’s evaluation is still in progress. The control nodes get these statuses and apply certain rules to choose the next node to be evaluated. In addition to having a hierarchical and modular structure, behaviour trees have the advantage of explicitly handling interruptions and failures.

Another popular technique is utility-based AI [65], which decides which action the AI agent should perform based on a scoring system. This concept implies processing data from within the game environment and evaluating all possible actions the agent can take.

A score is then assigned to each potential action depending on a specific utility function, such that the NPC can conduct the action with the highest score. The utility function is a core concept of this technique, as it gives each possible action a numerical value that reveals how adequate the respective action is in regards to the agent's goals. This approach is useful when creating agents that need to evaluate different options and choose the best one at any given moment.

Fuzzy logic [66] is also a decision-making technique that is relatively popular and used in several games. It is used to handle grey areas, which are situations that involve uncertainty or unpredictability regarding the manner in which a NPC takes decisions. This technique is based on the principle that concepts cannot be completely true or false, but instead they can be either one or the other to some degree. The system consists of fuzzy sets depicting traits of the AI agent, such as health or hunger, that are defined by either numerical or linguistic variables. Numerical variables range from 0 to 1, while linguistic variables are set by the developer and could indicate specific ranges, for instance "low", "medium", and "high". Depending on the state it is in, the character can perform certain actions.

A well-known and flexible AI technique is GOAP [67] which is used to create complex and adaptive NPC behaviour. In order to accomplish this, the developer needs to specify a set of actions and objectives from which the AI agent will be able to independently decide what actions will help achieving its goals. In this approach, any complex tasks that the agent needs to perform are simplified into a set of smaller tasks. The agent will evaluate them and choose those that can are most likely to reach its desired goals. This technique is particularly useful in games with dynamic environments and objectives, as it allows the AI agent to adapt to changes in real-time.

The decision-making systems discussed in this session are just a few of the many others that exist and are used in game development. However, no additional systems will be discussed in this paper, as the aforementioned ones are the most relevant to this project.

Having covered NPC behaviour management, it is now possible to delve into another critical area that enhances the realism of such agents. Pathfinding AI deals with calculating the most suitable paths that characters can take to move from one point to another within the game environment, while also avoiding certain obstacles. From the list of pathfinding algorithms that were discussed in the previous section, the A* algorithm is the one that will be given a closer examination. This technique is often used together with navigation meshes, which are abstract data structures that consist of 2D polygons depicting areas the NPC can traverse. Many libraries offer support for said structures, but game developers can also implement them on their own to fit their custom requirements. In the Vienna Game AI Library, the navigation mesh is built anew and designed as a 2D grid where its individual elements are referred to as nodes.

To understand how pathfinding was designed and implemented in the Vienna Game AI Library, it is important to examine the logic behind the A* algorithm. This technique requires as arguments two nodes from the navigation mesh. The first, also known as the start node, depicts the beginning point of the pathfinding search. The other, typically referred to as the end, goal, or target node, indicates the endpoint of the required path.

3. Background

During each cycle of its own main loop, the algorithm chooses a node that leads to the path with the lowest cost, which is calculated by using a heuristic function. This function sums up the total cost of the path from the start node to the current node and an estimation of the cost to reach the end node from the current node. The algorithm keeps track of all processed and unprocessed nodes, as well as the nodes chosen to construct the most optimal path such that it will be returned at the end. Depending on the size of the navigation mesh, the number of obstacles and the specifications of the device it is executed on, the A* algorithm may pose challenges in terms of performance and memory usage. The next chapters discuss how these risks are addressed in the solution given by the Vienna Game AI Library.

While pathfinding ensures NPC navigation across the game environment, in order to get naturally looking characters, they also need to move in a physically realistic manner, aspect that is achieved by using steering behaviours. These techniques have the same structure: they take data about their internal state and the state of the game world as input, then compute and return geometric data such as a velocity or direction to move in [11]. Games typically incorporate a large variety of such techniques, but this section will only cover the ones considered by and implemented in the proposed library.

One such technique is “seek”, which allows AI agents to steer towards a certain given target, continuously following it until it potentially reaches it. In case it catches up to its target, the agent will eventually pass through it due to moving at a constant speed, then turn back and continue to move back and forth. The “arrive” behaviour addresses this issue by forcing the character to slow down as it approaches the target, thus ensuring a smooth stop when reaching it. The “flee” behaviour is the inverse of “seek”, making the agent move in the opposite direction of a given target. Another useful technique is “pursue”, which is similar to “seek”, except that it involves predicting where the given target would be in the future. It then applies the “seek” behaviour to steer the agent towards the predicted location, giving it a higher chance to reach the target. “Evade” is the opposite of “pursue”, and it works on the same premise as it predicts the future location of the target, then uses “flee” to steer away from it. Sometimes, the AI agent will need to look at its target as it is approaching or pursuing it, and this is the behaviour known as “face”. So far, the techniques that were mentioned involve a given target that the agent interacts with in one form or another. There are also some that do not require such interactions, one being the “wander” behaviour. This technique deals with unpredictable NPC movement, which is useful in situations where characters need to move randomly within the game environment while maintaining a realistic and casual appearance.

In some cases, these steering behaviours are combined to enhance naturalism, as previously stated when referring to the “seek” and “arrive” behaviours. Another well-known combination of steering behaviours involves the “separation”, “alignment”, and “cohesion” behaviours. This combination forms the flocking technique, an algorithm that is used on multiple characters to simulate natural and realistic movement for groups such as herds of animals, flocks of birds, or swarms of insects. In this technique, the units are referred to as boids, a term introduced by Reynolds [13, 14]. Each of the three individual steering behaviours has an important role in ensuring this process performs smoothly.

3.2. Game AI Concepts

“Separation” is responsible for avoiding overlapping between boids, “alignment” guarantees maintaining an average velocity for the boids of the same group, and “cohesion” deals with centering the boids, keeping them close to their neighbours.

So far, the main AI domains targeted by the Vienna Game AI Library have been covered. There is one more area that has not been addressed, and although it is not part of the current plans for the library, it is worth mentioning as it has a big impact in game development. Learning AI refers to AI that adapts to the actions of the players by learning their techniques and skills, thus consistently providing a personalised experience. There are several underlying categories that form this domain, including action prediction, neural networks, reinforcement learning, and genetic algorithms. Action prediction, as the name implies, is about the AI agent figuring out the player’s next move. Neural networks consist of a large number of nodes that mimic brain cells and run the same algorithm in order to train the AI agent on specific tasks. Reinforcement learning is a technique through which NPCs learn how to make decisions based on trial and error, as they receive feedback on each action taken, thus adapting their behaviour to obtain the best results. Genetic algorithms are methods that simulate the natural evolutionary development for AI agents, allowing them to learn and evolve over generations.

While all AI techniques mentioned in this section offers their unique advantages, they can also pose risks and challenges that developers might face when integrating them into games. Firstly, AI is difficult to balance when attempting to ensure that the game experience is neither too easy nor too hard for the player. From a technical perspective, depending on implementation factors, AI may be computationally expensive, this potentially leading to a decreased game performance. Additionally, as game developers need to ensure that the AI agents maintain realism while adapting to the player’s actions and skills, they need to also be prepared and respond accordingly whenever the player acts in unpredictable ways.

With this section, it is expected that a general understanding of game and AI development was formed such that it is possible to dive into more theoretical aspects and explore how this library was created.

4. The Vienna Game AI Library

In this chapter, the Vienna Game AI Library is examined from various angles by delving into technical, structural, and functional aspects. The first section will cover the content and architecture of the library, while the second section will inspect the features of the library from a programming perspective.

4.1. Design

As stated in the preceding sections, the library is meant to be primarily used by students of the University of Vienna and so, in order to be easily integrated into other C++ projects and compatible with the collection of Vienna projects, it is written as a single-header file. This section begins by discussing the requirements that are essential for using the library, then continues with an overview of the AI domains and techniques chosen for implementation, concluding with an analysis of the general architecture of the library.

The first step before designing the library was to create it as a GitHub project. Hosting the project on GitHub ensures it is safely stored and easily accessible. Additionally, all its changes can be tracked and the project may be reverted to previous versions if needed. It also allows future students to work collaboratively on the project. To maintain consistency within the collection of Vienna projects, the library will be hosted on the supervisor professor's GitHub page [68].

When deciding what tools to use for building, packaging and compiling, the Vienna suite of projects was first examined. As one of the goals for the library was to integrate easily with the other projects, it made sense to continue using the same set of tools. To manage the build process, the CMake [69] tool was the preferred choice in the Vienna projects, hence it was adopted it by the proposed library as well. Regarding the build system and the compiler front end, it was decided to use Ninja [70] and Clang [71] as they are cross-platform and are known for significantly speeding up the build process, compared to other existing tools. They are very efficient in regards to large projects, which is an advantage for the Vienna Game AI Library as it is planned to expand over time. For generating documentation, similarly to the other Vienna projects, Doxygen [72] is used such that it generates a comprehensible Hypertext Markup Language (HTML) page from the code comments.

In order to simplify the process of building and running the library for developers, two batch scripts are provided. One is in charge of building the entire project, with the demo examples included, while the other simply runs the executable file. Furthermore, a step-by-step guide was created and is currently available in the README file [68]. At the time of writing, the proposed library is only configured for and tested on Windows.

Therefore, the guide and the batch scripts are relevant only for developers that use this operating system. In order to be able to run the demos and use the library, the developers first need to clone the project from GitHub. Then, they will need to ensure that the software mentioned above is installed and functional on their computer, as well as the relevant environment variables are set. Once all the requirements are met, the developers only need to run the batch scripts. Building the project will also generate the Doxygen documentation.

Since it was established from the beginning that the library will be written as a single header, all functionality exists in one file only. The library was implemented using the C++ 20 standard, similarly to the other Vienna projects. Before any plans were made on the structure of the library, a decision on which features the library should provide was made. After examining existing libraries such as the ones described in Chapter 2, in addition to checking what real-time strategy games usually require, three main areas were chosen for implementation: pathfinding, decision-making, and steering. These represent the minimum requirement for developing 2D RTS games, a fact that will be proved by the demos showcasing each feature.

Pathfinding is one of the requirements to create realistic NPCs in video games [73], as it ensures NPCs navigate seamlessly through the game world. Whether they need to chase or run away from something, to wander while preserving realism, or simply to get from one place to another, they need to find the most suitable route from their current position to their target. Pathfinding algorithms are used to calculate such paths, and in game development there are many that can be used depending on the requirements of each game. Strategy game worlds are typically large and dynamic, filled with obstacles that could change over time, thus requiring algorithms that can adapt easily to this dynamic characteristic while preserving efficiency. Algorithms such as Depth-First Search or Breadth-First Search could potentially impact the game performance, hence they are not suitable for this genre. Between Dijkstra’s algorithm and the A* algorithm, two of the most widely used ones, it seems that the latter is more effective, especially if used with navigation meshes [74]. Moreover, in addition to finding the shortest path as Dijkstra’s algorithm does, the A* algorithm also guarantees retrieving the most optimal ones since it uses heuristics. Evidently, to ensure this, it is important to use appropriate heuristics that do not overestimate and lead to computing longer paths. Rafiq et al. in [73] state that the A* algorithm continues to be one of the most popular pathfinding techniques, being often chosen by game developers due to its accuracy and performance. This notion is reinforced by Cui et al. [5], who also highlight the popularity of this algorithm.

Due to the reasons discussed above, the A* algorithm was selected to be part of the proposed library. As stated by Zikky in [74], the algorithm delivers even better results when combined with navigation meshes, factor that led to adopting both methods into the library. However, the algorithm can pose several challenges in terms of performance due to its time-consuming nature, especially on large navigation meshes. To address such challenges, an optimised version of the standard pathfinding technique was created. One of the introduced improvements focuses on decreasing the risk of increased CPU usage, objective that can be achieved by following the geometric preprocessing approach.

4. The Vienna Game AI Library

Algorithms such as A*, Landmarks and Triangle inequality (ALT) [75] and A*, Landmarks, and Polygon inequalities (ALP) [76] are part of this technique, in addition to graph simplification. The latter is the most suitable option for this project, as it can be used directly on the library’s custom navigation mesh. This process divides the navigation mesh into regions, then for each node it computes and stores the shortest path to each region. These calculations are done before the game starts such that at runtime the precomputed distances between regions can be retrieved with minimal time complexity. This process helps in ensuring low computational overhead at runtime, but to optimise it further and decrease the preprocessing time, multithreading was added. The intensive work of calculating all distances is now offloaded to a user-customizable number of threads which reduce the computational time and process the entire navigation mesh much faster. The right number of threads depends heavily on the number of CPU cores available on the user computer, as well as other factors such as the available memory and operating system constraints.

Another relevant aspect of the strategy game genre is controlling the decision-making flow of any AI agent. This feature is essential in games where NPCs need to simulate intelligence, adapt to the environment and react to the player’s actions in a realistic manner. Most games use simple decision-making systems, namely state machines and decision trees [16], while others might require more complex systems, specifically behaviour trees, rule-based systems, and goal-oriented behaviour. All these systems were evaluated for potential implementation and it was decided to begin with the simple ones and in time add more complex ones. Therefore, currently only decision trees and finite state machines are supported. Regardless of their simplicity, they allow AI agents to process information about their internal state or the game environment, such that they could choose an action and carry it out. Decision trees are widely used because they are fast, easy to implement and understand [16], and provide versatility when constructed as n -ary trees rather than binary trees. When it comes to finite state machines, it is clear that many, if not almost every game, take advantage of this feature in one form or another to control NPC behaviour [77]. In [78], Yoon et al. combine these techniques to create improved game agents, further solidifying the decision to incorporate them into the library.

The final area selected for implementation is represented by steering behaviours. These refer to NPC movement towards a given target while maintaining a desired speed and potentially avoiding obstacles. In this paper, movement refers to the way non-player characters move throughout the game environment, thus AI-driven movement, and not movement of their individual body parts. As Reynolds states in [11, 12], steering implies path determination, while movement of character body parts would be characterised as locomotion. There are a multitude of algorithms which control such movement and in order to choose some to include in the library, the most popular ones were examined. With regards to individual behaviours, the “seek”, “flee”, “pursuit”, “evasion”, “arrival”, “face”, and “wander” techniques were chosen. As for coordinated group movement, the flocking algorithm is offered, in addition to the three individual behaviours that it consists of. All these behaviours will ensure that non-player characters simulate intelligent movement across the game environment, thus leading to an increased fun factor for the players.

The implementation of each algorithm mentioned above is covered in the following section. While these features are arguably sufficient to create a 2D real-time strategy game, they represent only the foundation for the Vienna Game AI Library. As the project will be retained within the university, it will continue to be worked on and improved by future students or researchers.

Once all features that the library should contain were selected, Unified Modelling Language (UML) diagrams were created to facilitate the implementation phase. These high-level diagrams outline the structure of the library and give an overview of all connections among each feature’s components. They were created using the Lucid software [79].

To demonstrate how pathfinding is intended to work, Figure 4.1 illustrates how the navigation mesh is constructed such that it can be used to retrieve optimal paths when enabling geometric preprocessing.

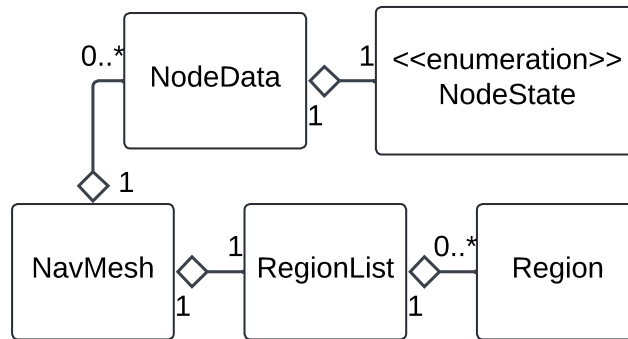


Figure 4.1.: UML Diagram for Pathfinding

The *NavMesh* class is responsible for creating the navigation mesh and setting any potential obstacles, as well as providing the functionality for geometric preprocessing and pathfinding. It consists of a user-customizable number of nodes, each of which is represented by the *NodeData* structure. This structure contains information relevant for the pathfinding algorithm, such as the *cost-so-far* and the heuristic value used to estimate the distance between the respective node and any other one. Each node has one state that can change throughout the course of the game and that can be either “Obstructable”, representing obstacle nodes through which game agents cannot pass, or “Walkable”, otherwise. If geometric preprocessing is used, the navigation mesh is divided into multiple regions, each being an instance of the *Region* struct. All of them are managed by the *RegionList* struct which stores them such that they can be iterated through during this process. Each *Region* struct is assigned a set of nodes which is calculated when initialising the navigation mesh. The *NavMesh* class has a *RegionList* instance, in addition to multiple *NodeData* objects. It provides methods to compute pathfinding with or without multithreading, as well as a function that starts the geometric preprocessing step. Additionally, it allows the user to save the structure of the navigation mesh to a file, load it from a file path, and offers options to manage the nodes within the mesh.

4. The Vienna Game AI Library

The decision-making flow is represented by Figures 4.2 and 4.3. The former illustrates the UML diagram for decision trees, while the latter presents the diagram for state machines.

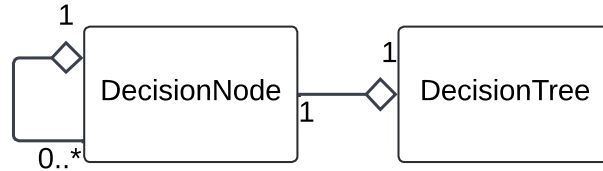


Figure 4.2.: UML Diagram for Decision Trees

The UML diagram for the decision tree, shown in Figure 4.2, is straightforward, as it consists of only two components. The tree itself has one root of type *DecisionNode*, which can have children, each of which can in turn have its own children. There is no limit to the amount of children a node can have, hence the decision tree is a *n*-ary tree. A *DecisionNode* instance contains methods for creating children and retrieving them based on a given index, while also storing all its children. The logic of each node, which is the condition to check in order to make a decision, relies on an abstract method that needs to be implemented by the user. During the game loop, the tree is traversed starting from the root node. Based on the decision made at the root node, the corresponding child node will be checked, and the process will continue in this manner until a leaf node is encountered.

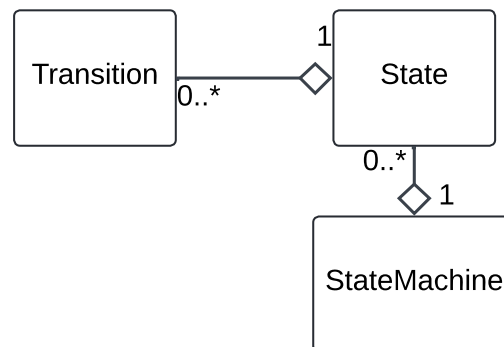


Figure 4.3.: UML Diagram for State Machines

The UML diagram for state machines depicts the underlying workflow process for this system in Figure 4.3. The *StateMachine* class acts as a manager of all states and transitions. It offers methods to create states and stores them such that it can iterate through them during the game loop. During each iteration of the loop, it checks which state needs to be activated and continuously tracks the current active one. The state machine can have multiple states, each of which can have multiple transitions.

The *State* class contains the actions that the AI agents need to perform when they are associated with it. It consists of three callbacks, namely *onEnterCallback()*, *onUpdateCallback()*, and *onExitCallback()*, which define variables or logic specific to the state. They are called either once the state is activated or deactivated, or repeatedly during the game loop. These methods are called by the *StateMachine* class as it is responsible for activating the states and ensuring they work as intended.

The purpose of the *Transition* class is to check certain criteria which would trigger the activation of other states. When constructed, they are linked to a certain state, which is also referred to as the target state, and are given a condition to check. When a condition managed by a transition is met, *StateMachine* activates the target state of the respective transition by calling its *onEnterCallback()*, and later *onUpdateCallback()*, functions.

Regarding movement, the UML diagram for the steering behaviours is shown in Figure 4.4. The diagram is intentionally simplistic, as it provides a basic overview of the key concepts required by this feature. For both individual and group movement, only two classes are needed, namely *Boid* and *Flock*.

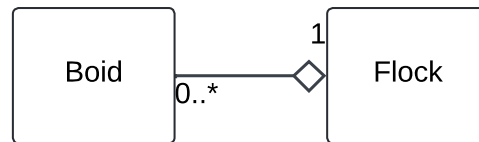


Figure 4.4.: UML Diagram for Steering Behaviours

The *Boid* class is named after the term introduced by Reynolds [13, 14] and is responsible for any type of steering from an individual point of view. Any non-player character that will require to steer in one form or another will need to be created as an instance of this class. This will allow using the techniques mentioned in the previous section, such as “seek”, “evade”, or “wander”. Moreover, the class contains methods for setting and retrieving information related to the agent, for instance its position, velocity, and speed.

While the *Boid* class contains the separation, alignment, and cohesion techniques that are used in the flocking algorithm, in order to manage the group movement easily, the *Flock* class was added. The purpose of this class is to create a group of boids, also referred to as a flock, and to call the algorithm on each individual boid. By using this class, it is easy to manage the group and ensure that all its members respect the rules of the algorithm. Additionally, the *Flock* class allows setting the ranges required for some of the individual steering techniques, while also storing all members of the newly created flock. The individual steering behaviours do not depend on this class, as they only exist with the *Boid* instance itself.

This section presented the design stage of the library by discussing its setup, structure, and content, while also providing the logic behind each choice made prior to the implementation stage. The following section will delve further into the technical side, as it will explore the previously mentioned classes and structs from a technical perspective.

4.2. Implementation

This section starts by detailing the implementation process of the Vienna Game AI Library, starting with custom data structures and data types and covering each technical aspect comprehensively. By the end of this unit, it is expected that the reader will have an extensive understanding of the logic behind the library's development.

Before diving into feature implementation, a set of data types and structures was chosen to ensure the efficiency and maintainability of the library. This stage is believed to be critical as it should prevent performance issues and bottlenecks if conducted properly. Various factors were examined, such as the memory the library might consume, suitable data structures for each individual feature, error-handling mechanisms, in addition to considering consistency and usability principles. Hence, it is believed that the selected data types and structures are relevant for the library's precision and performance. It is also important to note that all functionality is wrapped in a namespace to prevent naming conflicts and ensure compatibility with other projects.

In terms of data types, the library uses unsigned and signed integers, floating-point numbers, booleans, and strings. Each data type is thoroughly examined in this section.

Integers are data types that depict a range of mathematical numbers and do not include fractions, irrational numbers, or decimals. They can be unsigned, meaning that they only represent positive numbers, or signed, which depict both negative and positive numbers. In the library, the unsigned integers are used for countable objects, such as the number of children decision nodes, the width and height of the navigation mesh, or the index of an element in an array. Such numbers can never be negative, hence this data type is used to ensure that no exceptions or errors occur. The signed integers are only used in two instances: one in pathfinding, when the neighbours of a navigation mesh node are identified, and the other in a custom method that generates a random signed integer between a given range. The range of integer values varies depending on the bit size, meaning that a 64-bit integer has a bigger range. The library only uses 32-bit integers as they are sufficient for representing all values.

Floating-point numbers, or floats, represent real numbers, including decimals and fractions, and they are especially used for precise calculations and measurements. Thus, in the library they represent values such as velocity, speed, distances, and angles. There are two representations of the floating-point numbers, namely single precision and double precision. Typically, a single precision float uses 32 bits, while a double precision float uses 64 bits. The selection of the type is determined by the requirements of the application, and it usually depends if high accuracy is required, in which case double precision floats are the most appropriate choice. In game development, single precision floats are sufficient and effective as they allow for fast calculations without impacting the frame rate, while also requiring less memory than the double precision floats. For these reasons, the Vienna Game AI Library uses only single precision floating-point numbers.

In order to simplify these data types declarations and make the code cleaner, the *typedef* specifier was used to define *ui32* for 32-bit unsigned integers, *i32* for 32-bit signed integers, and *f32* for 32-bit floating-point numbers.

Booleans are data types that can have one of two values, typically denoted as true and false. They are used to control logic flow depending on certain given conditions. In the library, they are used in custom operators that compare two objects of the same class, as well as in controlling whether pathfinding uses preprocessing and multithreading.

Strings are data types that store text and are utilised when saving and retrieving a navigation mesh. This data type specifies the file path where the mesh should be located, as well as the path from which it should be retrieved.

The data types that have been covered so far are also known as primitive built-in data types. There are also user-defined types such as classes, structures, and enumerations. Classes and structures are used to construct specific objects such as *DecisionTree*, *Boid*, or *NavMesh*, consisting of different data types and structures, in addition to various methods. Enumerations ensure code readability as their purpose is to restrict a value to a range of constants. In the library, there is a custom *enum* declaration, namely *NodeState*, that holds two elements depicting the state of the NavMesh node. A node can be either “Obstructable” or “Walkable”, indicating whether the game agent can pass through it.

The library also incorporates various data structures, most of which are part of the standard C++ library. They are used to store and process data, while also providing efficient data access and modification. In the library, unordered maps (*std::unordered_map*), vectors (*std::vector*), and priority queues (*std::priority_queue*) are used. Vectors are the most preferred and used data structure in the library, as they are flexible and dynamic-sized containers. They store information such as the nodes in the navigation mesh, the shortest paths between regions, and the states and transitions of a state machine. In some cases, they even hold data structures such as unordered maps or other *std::vector* additional instances. Unordered maps are also used in pathfinding to store the paths between nodes and regions, as well as to retrieve them based on a given index. In addition to these data structures, the library also makes use of priority queues, but in one instance only. The A* algorithm requires a set which is commonly referred to as the “open set”, that stores all the nodes of the navigation mesh to be evaluated during the algorithm. Initially, the set contains only the start node but as the algorithm progresses, many other nodes will be added to it. During each iteration, the node with the smallest f value will be chosen from the set and evaluated, and depending on the size of this set, retrieving this node might pose challenges in terms of efficiency. Due to this reason, the open set is usually implemented using min-heaps or priority queue which would ensure that the node with the smallest f value is always at the top of the heap or queue. Thus, they allow for the respective node to be retrieved effectively. A choice was made to use priority queues over min-heaps as it is believed the former provides better code readability and maintainability.

The library also contains two custom data structures, namely *Vec2ui* and *Vec2f*. They represent custom 2D vectors of unsigned integers and floats, respectively. They contain arithmetic operations, such as addition, subtraction, multiplication, and division, as well as methods that allow printing them to the console or comparing them against other instances. These structures were created in order to ensure better control over their behaviour and performance, as they are tailored specifically to the needs of the library.

4. The Vienna Game AI Library

Vec2ui is used mostly to denote positions within the navigation mesh and uses unsigned integers as these do not need negative, decimal, or fractions representations. In situations where floating-point numbers are required, *Vec2f* is used, as seen when depicting velocity, positioning within screen coordinates, or any steering forces.

Having covered the data types and structures, the discussion can now proceed to feature implementation, starting with pathfinding. This implies discussing the A* algorithm, as well as the improvements added to the feature, namely the preprocessing step and the multithreading option. The research presented in Chapter 2 highlighted why the algorithm might benefit from being used together with navigation meshes. For those reasons, the Vienna Game AI Library includes a custom NavMesh class which is implemented as a 2D abstract grid that can be mapped to any 2D space. The class responsible for creating a NavMesh instance contains the necessary functions used in geometric preprocessing, pathfinding, and multithreading, along with several helper methods that are used for managing its content. The nodes of the navigation mesh are laid out in a one-dimensional array in a row major order to ensure data locality. If the array elements are stored next to each other, they can be accessed much more efficiently than if they were scattered across the memory. This leads to faster array iteration and node processing. While the nodes are stored in a 1D array, they are read in a 2D fashion as it facilitates code readability. Each node has between three and eight neighbours which are calculated during the navigation mesh initialisation. This implementation differs from others that only consider the top and bottom neighbours, as the library evaluates all surrounding nodes, including diagonal ones. The class also provides methods for retrieving a node by a given index, the height and width of the navigation mesh, as well as the index and the 2D coordinates of a node.

The A* algorithm implemented in the Vienna Game AI Library is very similar to the standard version that is commonly used, with several small modifications. Algorithm 1 presents the pseudocode for the A* implementation in the library. The algorithm takes as arguments the positions of the start and target nodes, along with a list of navigation mesh nodes. The start and target positions define the beginning and final points of the search. The resulting array, which is a *std::vector* of *Vec2ui* elements, represents the shortest path between the two given positions. If no path was found, the algorithm will return an empty array. The first step is to create the *parents* array, which will store the positions of each node that comprise the path, as well as the *openSet* priority queue, that will track all nodes that need to be evaluated during the A* search. Then, the *g* and *h* values of each node from the navigation mesh are initialised to *infinity* to ensure proper node evaluation. In situations where the algorithm is used multiple times during the game loop, these values need to be reset such that they do not retain any information from the previous computed paths. The *g* component is the cost from the start node to the current node, while the *h* value represents the heuristic estimate of the cost from the current node to the target node. Once they are set, the node with the start position is identified and its *g* value is set to 0, since it is the initial point of the search and the cost to reach itself is 0, then added to the open set. The main part of the algorithm begins once there are nodes in the priority queue and will loop through it as long as it is not empty. During each iteration, the top element of the priority queue is removed and evaluated.

If the respective element is the target node, the path is then computed by retrieving the nodes from the *parents* array, being returned in reverse order. Otherwise, the neighbouring nodes are identified, and for every neighbour that is not obstructed, the distance between the current node and the neighbour node is estimated. This estimation is determined by a heuristic function and the library offers two commonly used options, namely the Euclidean and the Manhattan distances.

```

Data: startPosition, targetPosition, nodes
Result: shortest path from startPosition to targetPosition
initialization of array parents;
initialization of priority queue openSet;
foreach node in nodes do
    | node.g  $\leftarrow \infty$ ;
    | node.h  $\leftarrow \infty$ ;
end
startNode  $\leftarrow$  node with startPosition;
startNode.g  $\leftarrow$  0;
openSet  $\leftarrow$  startNode;
while openSet not empty do
    | currentIndex  $\leftarrow$  openSet.pop();
    | if currentIndex = targetNodeIndex then
        | while currentIndex not -1 do
            | | shortestPath  $\leftarrow$  nodes[currentIndex];
            | | currentIndex = parents[currentIndex];
            | end
            | return shortestPath;
        | end
    | neighborIndices  $\leftarrow$  neighbor indices of node at currentIndex;
    | foreach neighborIndex in neighborIndices do
        | | neighbor  $\leftarrow$  nodes[neighborIndex];
        | | if neighbor.state = Obstructable then
        | | | continue;
        | | end
        | | tentativeG  $\leftarrow$  estimated distance to neighbor;
        | | if tentativeG < neighbor.g then
        | | | update neighbor g and h values;
        | | | openSet.push(neighbor);
        | | end
    | end
end
return empty array;

```

Algorithm 1: The A* Algorithm: Pseudocode

4. The Vienna Game AI Library

Both functions compute the distance between two given points in space, with their differences residing in the formula used in their calculations. The formula for the Euclidean distance is $d = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$, while the Manhattan distance is $d = |x_1 - x_2| + |y_1 - y_2|$, with (x_1, y_1) and (x_2, y_2) representing the coordinates of the two given points in a 2D space. The Manhattan function allows for horizontal and vertical movement, while the Euclidean function also supports diagonal movement. The choice of the heuristic function depends on the game environment, as well as the requirements of the game. Once the estimated distances to each neighbour are computed, the neighbour with the lowest g score is chosen and added to the *openSet* queue. Its parent, the previously evaluated node, is appended to the *parents* array.

As stated in the previous chapters, this algorithm might pose challenges depending on the size of the navigation mesh. The larger the navigation mesh is, the more calculations will need to be performed, factor that might impact the performance at runtime. To address this issue, geometric preprocessing is used, a technique that computes information about the navigation mesh before running the search algorithm. Also known as graph simplification, it is responsible for dividing the navigation mesh into large regions, leading to a more efficient pathfinding. In the library, the developer has the possibility to use pathfinding with and without this step. When the navigation mesh is constructed, it can take as arguments two values representing the amount of regions required on the X and Y axes, respectively. If none is given, the default value is 5 for both axes. This value was chosen in relation to the NavMesh size used in the demo example. After the initialisation of the nodes and their neighbours, the navigation mesh is divided into the given number of regions. A *regionID* is assigned to each node, depicting the region it is part of, while each *Region* instance stores the indices of its own nodes. These steps are done whether the preprocessing step is enabled. To use the technique, the *preprocess()* method can be called, preferably before the game loop in order to reduce computation time during the actual search. This method iterates over each region and calculates the shortest distance between each node of the navigation mesh and each region using the A* algorithm. The computed distances are stored in an array of unordered maps, each of which stores the shortest path from a NavMesh node to a specific region. When the computation is finished, the game loop can be initiated, allowing pathfinding to be executed with reduced impact on performance.

The *findPath()* and *findPreprocessedPath()* functions are responsible for finding the most optimal path between two given positions. The former simply calls the A* algorithm on the two nodes, while the latter is called only if geometric preprocessing was previously conducted. To find a preprocessed path, the method first identifies the nodes from the navigation mesh that have the same positions as the given arguments. If they are part of the same region, it simply calls the A* algorithm to retrieve the shortest path between them. If they are in separate regions, it retrieves the shortest path from the start node to the region where the target node is located. If the newly found path's end node is the target node, the path is then returned. Otherwise, the A* algorithm will be called to find the shortest path between this end node and the target node and return it if found. In the scenario that no paths are found, an empty array will be returned.

When retrieving a path between two nodes from the same region, or when retrieving the pre-calculated path between a node and a region, the function guarantees returning the shortest possible paths. However, in the case of invoking an additional A* call within a region, the path that is returned is referred to as the most optimal one, as it is not ensured that it is also the shortest. This can be observed in Figure 4.5, where the red square is the start node, and the blue square is the target node. The red line depicts the path computed after the preprocessing step, while the green one represents the path returned after calling only the A* algorithm. The node where the two lines get separated is the end node of the path found from the start node to the target region. Since it does not coincide with the target node and the A* algorithm is called within the region, the computed result is longer than the one retrieved by the other option.

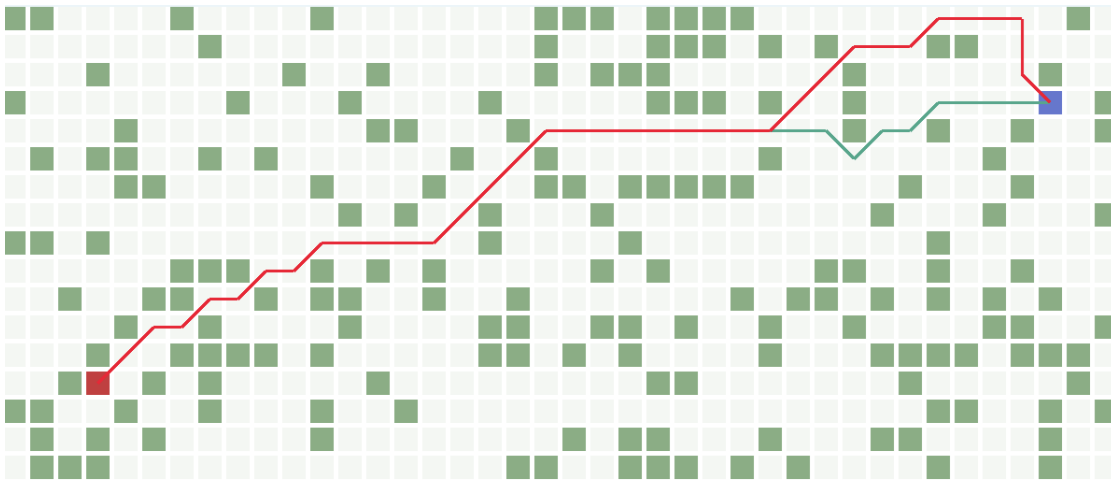


Figure 4.5.: Geometric Preprocessing vs. Standard A* Algorithm

While the preprocessing step optimises runtime performance, it still requires a significant amount of time during the calculations phase. Before discussing how this issue was addressed, it is important to note that the user-customizable size of a region can impact the performance if not chosen properly for the navigation mesh. Smaller regions could slow down the search, while increasing the chance of getting the shortest possible path. On the other hand, larger regions could lead to less accurate paths, but might have a smaller performance impact instead. Regardless of the way in which the regions are defined and created, geometric preprocessing is difficult to use in an efficient way, especially for large navigation meshes.

To enhance the performance of this technique, multithreading is used to perform the computation process in parallel, which is done by using the *thread* class from the standard C++ library. Each thread will process multiple regions simultaneously, leading to a reduced overall computation time. The *preprocess()* function mentioned above takes a boolean as argument which indicates whether to enable multithreading. If the parameter is set to “true”, the method will first determine the number of regions each individual thread should compute, then will delegate the *preprocessWorker()* function to it.

4. The Vienna Game AI Library

Although this method is distributed across multiple threads, it follows the same preprocessing logic as the *preprocess()* function, with one exception. During this step, since the A* algorithm is called to find the shortest paths between each node and each region, the *g* and *h* values are dynamically updated as the search process progresses and the nodes are explored. By using threads, such modifications can lead to conflicts where multiple threads attempt to read or write to these values simultaneously. This is known as data racing, a condition that can cause inconsistency or data corruption, thus affecting the pathfinding calculations. To mitigate this risk, each thread is given a copy of the NavMesh nodes such that they do not share editable data. There are also synchronisation mechanisms that can be used to ensure that the access to shared values is controlled, but in this scenario, considering the amount of calculations that need to be done, it was decided to opt for duplicating the array of NavMesh nodes. The number of threads can be set by the developer, which allows for a customised level of parallelism. This number depends heavily on the amount of CPU cores, thus it is reliant on the user computer.

The aforementioned techniques represent a significant improvement to pathfinding. However, there are a few ideas that are worth mentioning as they could further optimise the entire process. Currently, the preprocessing step stores the distances only while the application is running. Upon restarting the application, the respective method needs to be invoked again in order to recalculate the distances, which is only suitable for situations where the navigation mesh is dynamically changing. However, if the structure of the mesh is not changing often, the option to save the calculated values might be beneficial for developers. In this manner, when the game starts, it can immediately load the pre-computed data. Additionally, as mentioned in Chapter 2, there are many variants of the A* algorithm that could be implemented to speed up the technique. More information on potential improvements and optimisations is given in Chapter 5.

The entire pathfinding process, along with its improvements, is demonstrated in a small example project. It was already introduced earlier, as Figure 4.5 offers a snapshot of the demo. This demonstration is simplistic, offering a navigation mesh and the option to choose a start and a target node by left and right clicking, respectively. Once these are set, the user can press “R” which will compute two paths, one by calling the A* algorithm, and the other by retrieving the preprocessed data. Although this example illustrates pathfinding in a simple manner for understanding the differences between the two approaches, the demo for the state machine incorporates the feature in a more complex scenario. It illustrates how the feature can be used in the development of RTS games, and is explained in more detail later in the section during the discussion of FSMs.

The code snippet shown by Listing 4.1 demonstrates how the pathfinding features are utilised in a practical example, while focusing on the main methods that are called from the library. *VGAIL* is the namespace of the library, through which its functions and variables can be accessed from other classes. The initialisation of the navigation mesh takes as arguments the width and height of the mesh, as well as an obstacle value that defines the maximum percentage of the nodes that can be in the “Obstructable” state. The path is first created as an empty array. The preprocessing step is called in the following line, enabling multithreading and setting the number of required threads to 4.

In the game loop, the method to retrieve the preprocessed path is called on the start and end node positions. The commented line shows how to call the A* algorithm without any of the discussed improvements.

```

1  VGAIL::NavMesh* navmesh = new VGAIL::NavMesh(300.0f, 250.0f, 30.0f);
2  std::vector<VGAIL::Vec2ui> path;
3  navmesh->preprocess(true, 4);
4  while (!WindowShouldClose())    /* Game loop */
5  {
6      path = navmesh->findPreprocessedPath(startPos, targetPos);
7      // path = navmesh->findPath(startPos, targetPos);
8  }

```

Listing 4.1: Pathfinding with VGAIL: Code Snippet

Besides pathfinding, the library also includes decision-making algorithms, essential in developing intelligent behaviour for non-player characters. As stated previously, the two systems that the library offers are decision trees and state machines. The following discussion will examine the implementation of decision trees and their respective demo in the Vienna Game AI Library.

The implementation of this feature involves two classes, namely *DecisionNode* and *DecisionTree*. The *DecisionNode* class manages the tree nodes, consisting of methods for adding child nodes, retrieving a child node based on a specified index, and obtaining the count of children for the respective node. It also stores the child nodes into an array, which is essential when traversing the tree. The most important method of this class is *virtual void makeDecision(float deltaTime) = 0*, an abstract function that defines the logic of the decision node. It needs to be implemented manually by the developer, and it should either contain a set of instructions for the AI agent, or delegate further work to its child nodes based on some criteria. Listing 4.2 provides a code snippet of the example created to showcase this system. The *IsGuardClose* class, which represents the root of the tree, forwards the next decision to its corresponding child based on the distance between two characters. Alternatively, the *IsCloseToCastle* node holds the logic depending on the given condition.

The *DecisionTree* class oversees the management of the tree, comprising several essential methods. It is responsible for creating the root node, from which the tree is then constructed. It also ensures that the tree is properly reset upon deletion. Its most important function, *update()*, performs the decision-making process by iterating through the tree and calling the *makeDecision()* function for the corresponding node based on the chosen decision. In the code snippet shown by Listing 4.2, line 25 demonstrates how the root is instantiated in a practical example, followed by the creation of its child nodes. Then, during the game loop, the tree is being updated in relation to the delta time, which is the time between two consecutive frames.

As outlined before, the system implemented in the library supports n -ary tree structures, allowing for multiple children nodes, rather than being limited to binary nodes. The reason behind this choice relies on the advantage it brings, namely the possibility to represent multiple decisions. Additionally, depending on how the tree is created, it can lead to faster traversal and evaluation as it might require fewer levels than a binary tree.

4. The Vienna Game AI Library

```
1  class IsGuardClose : public VGAIL::DecisionNode {
2      ...
3      void makeDecision(float dt) override {
4          float dist = VGAIL::distance(burglar->pos, guard->pos);
5          if (dist <= tileSize * 3.0f) {
6              getChild(0).makeDecision(dt);
7          } else {
8              getChild(1).makeDecision(dt);
9          }
10     }
11 };
12 class IsCloseToCastle : public VGAIL::DecisionNode {
13     ...
14     void makeDecision(float dt) override {
15         float dist = VGAIL::distance(burglar->position, castlePos);
16         if (dist >= tileSize / 4.0f) {
17             burglar->mode = BurglarMode::WALKING;
18         } else {
19             burglar->mode = BurglarMode::ENTERING;
20         }
21     }
22 };
23 int main(int argc, char* argv[]) {
24     VGAIL::DecisionTree burglar_tree;
25     VGAIL::DecisionNode& burglar_root = burglar_tree.createRoot<
26     IsGuardClose>(burglar, guard);
27     burglar_root.addChild<IsInDanger>(burglar, castlePos);
28     burglar_root.addChild<IsCloseToCastle>(burglar, castlePos);
29     while (!WindowShouldClose()) /* Game loop */
30     {
31         float deltaTime = GetFrameTime();
32         burglar_tree.update(deltaTime);
33     }
```

Listing 4.2: Decision Trees with VGAIL: Code Snippet

To demonstrate the use of this feature in a game-like scenario, a demo example was created in which two characters have their own decision trees. A snippet of the demo is shown in Figure 4.6. The example does not require any input, as the NPCs act on their own in relation to their decision-making system. The idea behind this demo is simple: a guard is defending a castle and a burglar is trying to break in. Both of them start with 3 lives and an action to perform immediately. The guard is always patrolling, unless the burglar gets too close, in which case they fight until either of them has one life remaining. The character that reaches this threshold first will run away, allowing the other to continue pursuing its goals. The decision tree for the guard is straightforward, as it requires minimal decision-making. The guard's behaviour is slightly more complex, as it also needs to check whether it can open the gate to the castle, as well as wait for a certain time to pass to succeed. Each decision that is taken by the characters is displayed on the screen in text format for a better overview of the process.

The decision tree demo is simplistic, but it offers insight into reasons why such a system is useful in games, especially in the real-time strategy subgenre. While the version included in the library is valuable, there are additional improvements that can be implemented. One aspect that would ease the implementation process for the developers is the option to save the tree structure to a file, such that it can be loaded upon necessity. The logic for the nodes would have to be manually set by the developer, but this option would construct the tree automatically without having to define each node relationship. Another idea is providing a user interface through which developers could create the decision tree. It would require mouse events such as “click” and “drop”, which would also save the developers from writing extensive lines of code. Finally, if the system, along with all the mentioned improvements, is not sufficient in relation to a game’s requirements, behaviour trees may be considered, as they ensure additional flexibility and modularity. Further information on the benefits they bring is given in the following chapter.

Guard health: 3

Burglar health: 3



Figure 4.6.: Demo for Decision Trees

While decision trees are a valuable tool in the decision-making area, state machines are employed to handle more complex non-player character behaviour. The library supports this technique for this specific reason. The system consists of three classes which control the entire decision-making flow. The *Transition* class is responsible for determining which state needs to be activated. Upon initialisation, the constructor takes two arguments, namely the target state and a callback method. The target state is the state for which the *Transition* instance evaluates conditions in order to decide on its activation. The callback is a function that can be passed as an argument and represents the condition that needs to be checked. This method needs to be implemented manually by the developer, as it is only declared within the class.

4. The Vienna Game AI Library

An instance of this class cannot be created on its own as this can be achieved through the *State* class by calling its *addTransition()* method. Listing 4.3, taken from the demo example, shows how this can be applied in a game-like scenario. Lines 6-8 display the mentioned function that takes as an argument a state and a callback method which checks whether a value meets the specified criteria. The *Transition* class does not contain other methods, as its logic relies on the callback method.

```
1  int main(int argc, char* argv [])
2  {
3      VGAIL::State* dropOffState = stateMachine.createState();
4      VGAIL::State* locateHomeState = stateMachine.createState();
5
6      locateHomeState->addTransition(dropOffState, [&]() {
7          return currentPathIndex == path.size() - 1;
8      });
9
10     locateHomeState->onEnterCallback = [&]() {
11         currentStateIndex = 1;
12         path = navmesh->findPath(workerPos, homePosition);
13     };
14
15     locateHomeState->onUpdateCallback = [&](float delta) {
16         if (currentPathIndex < path.size() - 1) {
17             // move towards home
18         }
19     };
20
21     locateHomeState->onExitCallback = [&]() {
22         path.clear();
23         currentPathIndex = -1;
24     };
25
26     while (!WindowShouldClose()) /* Game loop */
27     {
28         stateMachine.update(GetFrameTime());
29     }
30 }
```

Listing 4.3: State Machines with VGAIL: Code Snippet

The states of an AI agent are represented by the *State* class which contains three callbacks, and a couple of methods to manage its transitions. The function that adds transitions between this state and any other *State* instances has been covered above. Additionally, the class provides a method to retrieve all transitions associated with its instance, all of which are stored in an array. The callbacks are abstract methods, meaning they need to be implemented manually by the developer, and are invoked depending on the status of the state. *onEnterCallback()* is a method that is called when the state gets activated, while the *onExitCallback()* is triggered when the state is deactivated or exited. *onUpdateCallback()* is the third callback which is called continuously during the game loop, as long as the state is active. These functions contain the logic which the non-player character must follow while it is in the respective state.

The code snippet displayed in Listing 4.3 presents an example of how these methods can be implemented. Once the *locateHomeState* state is activated, it will trigger *onEnterCallback()* which will set the current state index to 1 and compute the path to the home's location. Then, for each iteration of the game loop while the state is active, *onUpdateCallback()* will be invoked to continuously move the character in the direction of its home. Lastly, upon state deactivation, the *onExitCallback()* method will clear the *path* array and another state may get activated.

To manage the workflow of this system, the *StateMachine* class was created. It consists of three methods which deal with creating, updating, and retrieving the states. Creating a state simply initialises a *State* object and adds it to an array. Updating them is possible through the method also seen on line 28 from Listing 4.3. Every iteration of the game loop, this class evaluates whether any transition has been triggered, in which case it deactivates the current state by calling its *onExitCallback()* method, then activates the new state by invoking its *onEnterCallback()* function. While the state is active, it updates it per frame until another transition is triggered. The class also allows for retrieving the current active state. It is important to note that, due to the fact that the number of states is always predefined, the state machine offered by the library is a FSM.



Figure 4.7.: Demo for State Machines

Figure 4.7 presents an example showcasing the potential usage of a state machine in a real-time strategy game. This demo does not require any user input as the AI agent performs actions on its own according to the state machine workflow. There are five possible states: collecting resources, locating and moving to the home's position, dropping off resources at home, locating the closest non-empty mine and moving towards it, and idling. The simulation starts with the character collecting resources at a specific mine.

4. The Vienna Game AI Library

There is a certain amount of resources that it can carry, so once this limit is reached, it goes to its home to drop off the baggage. Then, it locates the nearest non-empty mine and starts gathering resources once it is there. The process is repeated until all mines are empty. Once it locates the home or one of the mines, the character uses the pathfinding technique described earlier in this section to reach its destination. The game environment is laid out on the navigation mesh which facilitates the entire navigation process. The demo also displays the current state of the agent to illustrate its decision-making flow.

Similarly to the previous features covered in this section, there are potential optimisations regarding the state machine system. It was mentioned in Chapter 2 that a well-known improvement that is often considered is the hierarchical state machine. The hierarchical structure of HSMs leads to a simplified state management and reduces redundancy, which is useful in situations where the system can grow large and may contain numerous states and transitions. Another option is one that has been already discussed in the case of decision trees, namely saving the structure of the system to a file and loading it upon necessity. The logic of each component would still be written manually by the developer, but the relationships between them could be saved in a graph format, then retrieved to automatically reconstruct the system. Additionally, a user interface may be beneficial as it would allow developers to easily create, modify, and visualise different states and transitions.

With decision-making systems thoroughly explored, the next topic of discussion is steering behaviours. These behaviours are critical in RTS games as they are responsible for enabling non-player characters to move through the game environment in a realistic and smooth manner while avoiding obstacles. The library provides these techniques through the *Boid* and *Flock* classes. An instance of the *Boid* class is an AI agent that can use steering behaviours. It has a position, a velocity, and a maximum speed, all of which are defined during its initialisation. The class provides methods for retrieving and changing this information, along with the functionality for enabling any of the steering techniques. Below, each technique is separately examined, but it is important to highlight two methods that are used to update the *Boid* instance once the steering methods are invoked. One of them is *applySteeringForce()*, a method that applies the newly calculated steering force to the boid's velocity. The other, *updatePosition()*, is used to update the position of the *Boid* instance in relation to the time difference between frames. Both functions need to be called once any steering force is calculated, otherwise no changes will be registered. Listing 4.4 illustrates how these methods can be used with any steering technique to adjust the NPC's movement.

```
1 VGAIL::Boid* agent = new VGAIL::Boid(startPosition, startVelocity,
2   maxSpeed);
3 VGAIL::Vec2f steeringForce = agent->seek(playerPosition,
4   maxAcceleration);
5 agent->applySteeringForce(steeringForce);
6 agent->updatePosition(deltaTime);
```

Listing 4.4: Steering Behaviours with VGAIL: Code Snippet

The “seek” behaviour allows non-player characters to find the direction of a given target and to move towards it. Listing 4.5 presents the implementation of this technique within the library. The function takes as arguments the position of the target and a maximum acceleration. By subtracting the agent’s position from the target’s position, it computes the direction from the agent to the target. The resulting vector is then normalised, such that it maintains a consistent magnitude which is critical in ensuring uniform steering. Finally, the vector is multiplied by the given maximum acceleration which guarantees that the character does not accelerate indefinitely. The final result can then be applied to the character by invoking the two methods previously mentioned.

```

1  Vec2f seek(Vec2f targetPosition, f32 maxAcceleration)
2  {
3      Vec2f steeringForce = targetPosition - m_position;
4      steeringForce.normalize();
5      steeringForce = steeringForce * maxAcceleration;
6      return steeringForce;
7  }

```

Listing 4.5: “Seek” Behaviour: Code Snippet

The “flee” behaviour closely resembles the “seek” behaviour and follows the same logic, with one key difference. Instead of moving towards a given target, this technique allows the non-player character to move away from it. From the programming perspective, the key difference lies in the line 3 from Listing 4.5. The subtraction between the two positions is reversed, enabling the character to move in the opposite direction of the target.

There is one issue that is caused by the “seek” behaviour and addressed by the “arrive” behaviour. The technique will not present challenges if the target moves at a higher speed than the AI agent, which will continuously seek it. The problem arises when the target stops moving or is slower than the agent, in which case the agent has the possibility to reach it. When attempting to arrive at the target, the “seek” behaviour can make the agent overshoot the target due to its constant speed, causing it to reverse the direction of moving. The agent will keep going back and forth, leading to oscillating around the target indefinitely. This issue is addressed by the “arrive” behaviour, which lets the character slow down as it is approaching the target such that it can stop smoothly at the right location. Listing 4.6 displays how this is performed in the library. The method that enables this technique takes as arguments the position of the target, a slow radius and a maximum acceleration. The slow radius is responsible for slowing down the non-player character once it is crossed. The first calculation determines the direction to the given target, which is then evaluated based on its length. If the magnitude of this vector is less than a certain threshold, it means that the character has reached the target and it can stop moving. Then a suitable speed for the character is calculated depending on its position within the slow radius. If the agent has not yet crossed it, it can move at full speed. Otherwise, the speed is calculated according to the distance from the target, decreasing progressively as the character approaches it. The current velocity of the character is then subtracted from the newly calculated vector which computes the final steering force. Before returning it, the algorithm checks whether it needs to scale down the steering force based on the maximum acceleration.

4. The Vienna Game AI Library

```
1   Vec2f arrive(Vec2f targetPosition, f32 slowRadius, f32
2   maxAcceleration)
3   {
4       Vec2f desiredVelocity = targetPosition - m_position;
5       f32 distance = desiredVelocity.getMagnitude();
6
7       if (distance <= 0.01f)
8           return Vec2f{};
9
10      desiredVelocity.normalize();
11
12      if (distance > slowRadius) {
13          desiredVelocity = desiredVelocity * m_maxSpeed;
14      } else {
15          desiredVelocity = desiredVelocity * m_maxSpeed * distance /
16          slowRadius;
17      }
18
19      Vec2f steeringForce = desiredVelocity - m_velocity;
20
21      if (steeringForce.getMagnitude() > maxAcceleration) {
22          steeringForce.normalize();
23          steeringForce = steeringForce * maxAcceleration;
24      }
25
26      return steeringForce;
27 }
```

Listing 4.6: “Arrive” Behaviour: Code Snippet

When seeking a moving target, the AI agent will continuously navigate towards its position at that specific moment. Upon reaching this position, the target will have moved. This could pose a challenge in situations where the distance between the agent and the target is considerably long as it will not be possible for the agent to chase the target in an intelligent manner. The “pursue” behaviour addresses this issue by relying on predictions. It makes an assumption on the target’s movement and estimates where it will be in the future, such that the NPC can move towards that position. Listing 4.7 shows how this feature is executed in the library. The *pursue()* function requires three arguments, namely the target itself as a *Boid* instance, a maximum acceleration and a maximum prediction. It first calculates the distance between the character and the target. If the distance is below a certain threshold, it is implied that the character has reached the target and does not need to continue moving. Otherwise, the prediction step takes place. The algorithm calculates the *prediction* value, which is the predicted time to reach the target’s current position at maximum speed. If the speed is too small or the distance to the target is too large, the predicted time can get considerably large, which is not a practical option. To limit it, the maximum prediction given as an argument to the function is used. The newly computed predicted time is then used to calculate the future position of the target if it maintains its current velocity. As the estimation part is finished, the “seek” technique can be applied by passing the result as the target position to the respective method.

```

1  Vec2f pursue(const Boid* target, f32 maxAcceleration, f32
maxPrediction)
2  {
3      f32 speed = m_velocity.getMagnitude();
4      Vec2f direction = target->getPosition() - m_position;
5      f32 distance = direction.getMagnitude();
6
7      if (distance <= 0.01f)
8          return Vec2f{};
9
10     f32 prediction;
11     if (speed <= distance / maxPrediction)
12         prediction = maxPrediction;
13     else
14         prediction = distance / speed;
15
16     Vec2f newPosition = target->getPosition() + target->getVelocity()
* prediction;
17     return seek(newPosition, maxAcceleration);
18 }

```

Listing 4.7: “Pursue” Behaviour: Code Snippet

The “evade” behaviour works similarly to the “pursue” behaviour, with one minor difference. As this technique allows the agent to move away from the target based on predictions, instead of using the “seek” behaviour on the newly computed position, it invokes the “flee” method. Given that the distinction between the two behaviours is represented by just one line, a code snippet for the technique is not provided such that redundancy is avoided.

The implementation of the “face” behaviour differs from the ones discussed above, as it does not return a steering force. This technique ensures that the non-player character always looks in the direction it is moving, leading to realistic and natural motion. Listing 4.8 demonstrates the logic behind the method that offers the “face” technique. The *Boid* class contains one variable that is relevant for this function, namely *m_lastRotation* which depicts the last rotation stored for this *Boid* instance. If the velocity of the boid is less than a given threshold, meaning that it is not moving, it returns its last saved rotation. Otherwise, it computes the direction in which it is moving, then calculates the angle in radians from the current position of the boid to the target location.

```

1  f32 getRotationInDegrees()
2  {
3      if (m_velocity.getMagnitude() < 0.01f)
4          return m_lastRotation;
5      Vec2f target = m_position + m_velocity;
6      f32 radians = std::atan2(target.y - m_position.y, target.x -
m_position.x);
7      m_lastRotation = radians * (180.0f / PI);
8      return m_lastRotation;
9  }

```

Listing 4.8: “Face” Behaviour: Code Snippet

4. The Vienna Game AI Library

The result, converted to degrees, can be used when rendering the character, as it can be seen in Listing 4.9. This code snippet illustrates the texture drawing of a character using the Raylib library. The *DrawTexturePro()* function takes as one of the arguments the rotation of the texture, which is shown on line 6. By using the functionality offered by the library, the character will always be rendered facing the direction of its movement. In this specific example, a 90-degree offset is applied to the result in order to align the texture correctly.

```
1 DrawTexturePro(  
2     dogTexture,  
3     { 0.0f, 0.0f, dogTexture.width, dogTexture.height },  
4     { dog->getPosition().x * tileSize, dog->getPosition().y *  
tileSize, 30.0f, 35.0f },  
5     Vector2{ 15.0f, 17.5f },  
6     dog->getRotationInDegrees() - 90.0f,  
7     WHITE  
8 );
```

Listing 4.9: “Face” Behaviour with VGAIL: Code Snippet

The “wander” behaviour enables non-player characters to navigate in a random manner throughout the game environment, while preserving realism. To understand the logic behind its implementation, it is important to discuss the fundamentals of this behaviour. Following Craig Reynolds’s proposal, an imaginary circle is placed in front of the character, acting as a guide for the character’s direction of movement. During each frame, a point is randomly chosen along the circle’s edge such that it creates a new target direction. Thus, as the character follows the target point which changes subtly every frame, it smoothly navigates within the game world without taking any sudden or unnatural turns. Figure 4.8 demonstrates this logic by presenting the imaginary circle, along with its radius and target point that is chosen within a displacement range. The triangle acts as the AI agent with a direction of movement that changes every frame.

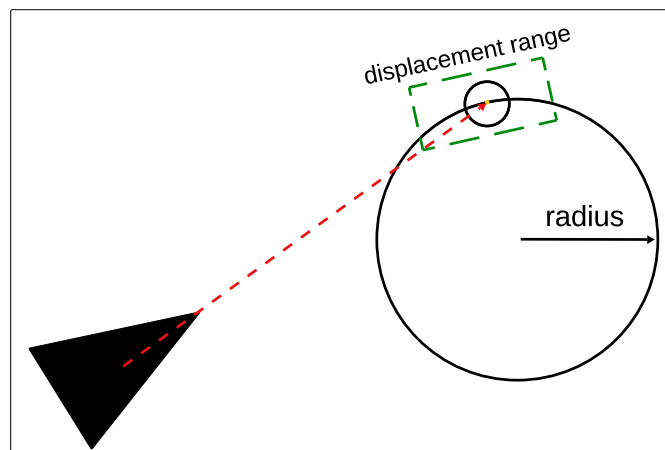


Figure 4.8.: Wander Behaviour Diagram

The implementation of this technique within the Vienna Game AI Library is depicted by Listing 4.10. The function that offers this technique requires four arguments. The *circleDistance* variable depicts the distance between the character and the imaginary circle, while *circleRadius* represents the size of the circle. The *displacementRange* value is the range from which the random point is selected. As for the previous techniques, *maxAcceleration* is the maximum rate at which the character can change its velocity. The function evaluates whether the character’s velocity is under a certain threshold, such that it returns an empty *Vec2f* object if the character is not moving. Otherwise, it computes a vector by copying the character’s velocity and setting its length to the distance between the character and the imaginary circle. Lines 10-12 calculate the coordinates of the randomly chosen point along the circle’s edge, where *m_theta* is a member variable of the *Boid* class that is updated regularly with a random value between the displacement range to ensure aimless movement. Finally, the function returns the direction in which the character needs to move based on the newly calculated random point.

```

1   Vec2f wander(f32 circleDistance, f32 circleRadius, f32
2   displacementRange, f32 maxAcceleration)
3   {
4       if (m_velocity.getMagnitude() <= 0.01f)
5           return Vec2f{};
6
7       Vec2f desired = m_velocity;
8       desired.setMagnitude(circleDistance);
9       desired = desired + m_position;
10
11      f32 theta = m_theta + getRotationInDegrees();
12      f32 x = circleRadius * cos(theta);
13      f32 y = circleRadius * sin(theta);
14
15      desired = desired + Vec2f{x, y};
16      Vec2f steeringForce = desired - m_position;
17      steeringForce.normalize();
18      m_theta = m_theta + randomFloat(-displacementRange,
19      displacementRange);
20
21      return steeringForce * maxAcceleration;
22  }

```

Listing 4.10: “Wander” Behaviour: Code Snippet

Each of the steering techniques discussed above has its own demo, all interactive with the exception of the “wander” example. For the other behaviours, simplistic scenarios were created where the user controls the movement of an object, leading to the AI agent seeking, pursuing, evading, arriving at, or fleeing from it. The example depicting the “wander” behaviour consists of one AI agent that has the respective force applied to it, therefore it does not require any user input. These smaller projects primarily help developers grasp how the techniques work individually.

To demonstrate the usage of all these behaviours combined in a RTS game, a more complex example was created. A snapshot of this demonstration is displayed in Figure 4.9.



Figure 4.9.: Demo for Steering Behaviours

In this scenario, there are multiple AI agents, each with their own role. During the day, the chickens and the dog move around the game world using the “wander” technique. At night time, the chickens use “arrive” to move towards the barn, as well as to smoothly stop there once they reach it. During the night, a snake spawns and wanders until it gets close to a chicken, in which case it follows it by using “seek” to attempt catching it. The dog is aware of the snake and uses the “pursue” technique to seize it. If caught, the snake disappears and another one spawns if it is still night time. The snake does not see the dog until the distance between them is equal to or smaller than a given threshold, in which case the snake will use the “evade” behaviour to escape the dog. All characters in this example also face the direction in which they are moving.

The flocking algorithm is the last technique included in the library. As previously stated, it consists of three steering behaviours, namely “separation”, “alignment”, and “cohesion”, each of which has a critical role in enabling the group movement. They are also provided by the library in separate forms, not only as part of the flocking algorithm, for situations where they might be needed individually.

The “separation” behaviour ensures that the members of the flock do not overlap, thus it steers them away from one another based on each boid’s own separation range. This range defines the distance at which a boid needs to stay away from its neighbours, such that there is space between them. Listing 4.11 shows the implementation of this technique, thus offering a clearer overview of its functionality outside of the flocking algorithm.

The method iterates through the flock and computes the distance between the current boid and the other members. If any boids are within the separation range, the current boid steers away from them with a force determined by *avoidFactor*, the value provided as an argument.

```

1  Vec2f separation(f32 separationRange, f32 avoidFactor, const std::
   vector<Boid*>& flock)
2  {
3      Vec2f separationVector;
4
5      for (Boid* other : flock)
6      {
7          f32 dist = distance(m_position, other->getPosition());
8          if (other->getID() != m_id && dist < separationRange)
9              separationVector = m_position - other->getPosition();
10     }
11
12     return separationVector * avoidFactor;
13 }

```

Listing 4.11: Separation Behaviour: Code Snippet

“Alignment” or “Align” is the behaviour that calculates the average velocity of the flock and steers the member accordingly. Listing 4.12 illustrates the implementation within the Vienna Game AI Library. Instead of a separation range, this technique requires a perception range within which boids consider other boids as their neighbours. For a given boid, it calculates the average velocity of its neighbours and computes the steering force. This force depends on the *matchingFactor* variable, which depicts how strong the given boid needs to steer such that it matches the velocity of its neighbours.

```

1  Vec2f align(f32 perceptionRange, f32 matchingFactor, const std::
   vector<Boid*>& flock)
2  {
3      Vec2f alignVector;
4      ui32 neighbors = 0;
5
6      for (Boid* other : flock)
7      {
8          f32 dist = distance(m_position, other->getPosition());
9          if (m_id != other->getID() && dist < perceptionRange)
10         {
11             alignVector = alignVector + other->getVelocity();
12             neighbors++;
13         }
14     }
15
16     if (neighbors > 0)
17         alignVector = alignVector / neighbors;
18
19     return (alignVector - m_velocity) * matchingFactor;
20 }

```

Listing 4.12: Align Behaviour: Code Snippet

4. The Vienna Game AI Library

Lastly, “cohesion” steers the boids towards the average position of the group they are part of, as shown in Listing 4.13. The perception range serves the same purpose as in the “align” technique, which is to identify the neighbours of a given boid. The average position of all group members is calculated such that the boid can be steered towards the centre of the group. The force which it is steered with depends on *centeringFactor*, a floating-point value given as an argument to the function.

```
1   Vec2f cohesion(f32 perceptionRange, f32 centeringFactor, const std::
vector<Boid*>& flock)
2   {
3       Vec2f cohesionVector;
4       ui32 neighbors = 0;
5       for (Boid* other : flock) {
6           f32 dist = distance(m_position, other->getPosition());
7           if (m_id != other->getID() && dist < perceptionRange) {
8               cohesionVector = cohesionVector + other->getPosition();
9               neighbors++;
10          }
11      }
12      if (neighbors > 0)
13          cohesionVector = cohesionVector / neighbors;
14      return (cohesionVector - m_velocity) * centeringFactor;
15  }
```

Listing 4.13: Cohesion Behaviour: Code Snippet

Having discussed these techniques individually, it is possible to examine how they are used together to create the flocking algorithm. The class responsible for this technique is the *Flock* class, which manages all members of the flock. It provides functionality for setting the separation and perception ranges that are essential for the three steering techniques, in addition to storing the boids such that they can be iterated through. Its main method, *update()*, invokes each boid’s *doFlocking()* function that combines the three techniques discussed previously. Listing 4.14 shows an example of using the algorithm, depicting all methods that must be called to enable this feature. Evidently, the *update()* method needs to be called within the game loop such that it continuously updates the position and velocity of each boid.

```
1   int main(int argc, char* argv[])
2   {
3       VGAIL::Flock* flock = new VGAIL::Flock();
4       flock->setRanges(separationRange, perceptionRange);
5       for (uint32_t i = 0; i < 100; i++) {
6           flock->addBoid(position, velocity, minSpeed, maxSpeed);
7       }
8       while (!WindowShouldClose())    /* Game loop */
9       {
10          flock->update(deltaTime, avoidFactor, matchingFactor,
centeringFactor);
11      }
12  }
```

Listing 4.14: The Flocking Algorithm with VGAIL: Code Snippet

The example created to showcase the functionality of this algorithm consists of a group of AI agents that simulate a swarm of fish. A simplistic user interface is also provided, allowing the user to modify the three mentioned factors, namely avoid, matching, and centering, such that their influence on the three steering behaviours can be observed. A snapshot of this demonstration can be seen in Figure 4.10. A custom function implemented specifically for the demo example ensures that the boids stay within the window boundaries.

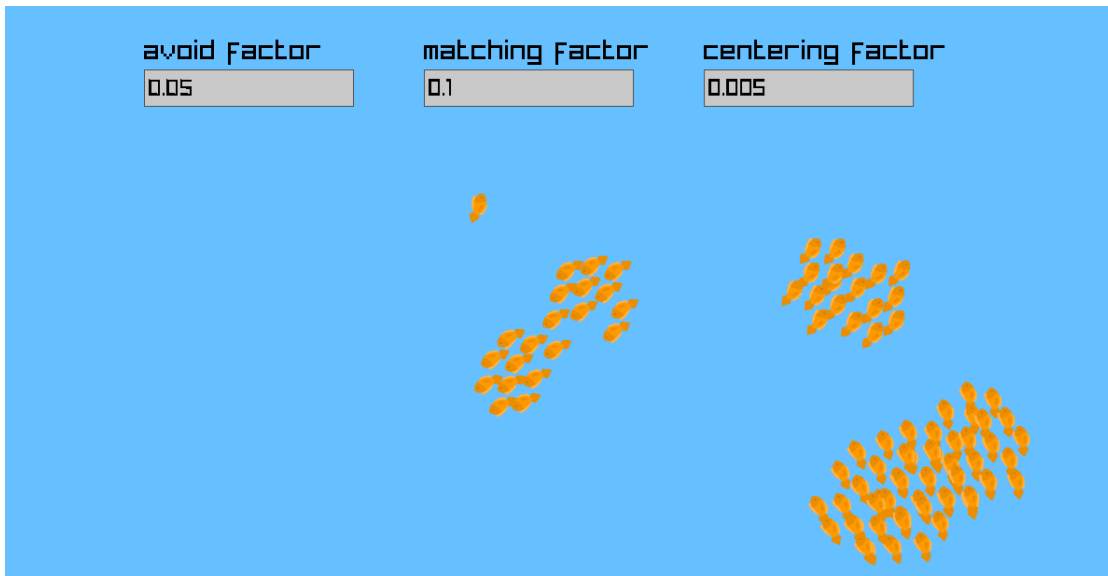


Figure 4.10.: Demo for The Flocking Algorithm

It is important to reference the materials that aided in the development of the features presented in this chapter. As stated throughout the thesis, the steering behaviours were implemented following Reynolds’s proposals [11, 12, 13, 14]. Moreover, for several individual techniques, [80, 81, 82, 83] served as inspiration during the implementation phase. The “Boids algorithm - augmented for distributed consensus” article [84] helped during the implementation of the flocking algorithm. The pseudocode offered in [85] also acted as a guide when developing pathfinding. Lastly, the “Artificial Intelligence for Games” book [16] provided guidance, especially when implementing the steering techniques, but also during the development of decision trees and state machines. The demo examples that showcase each feature were created using different textures [86, 87, 88, 89, 90, 91, 92] and fonts [93]. In the flocking example, [94] also helped in creating the user interface. All code snippets and figures offered in this chapter are part of the Vienna Game AI Library and its demo examples. Some are slightly adjusted to fit the explanations in a clear and concise way.

This chapter outlined the setup, architecture, and content of the library, along with the details of each implemented feature and its respective demo. Having covered the technical part of this project, the next chapter will focus on evaluating the library from three key perspectives, analysing the results, and discussing further steps and improvements.

5. Evaluation and Discussion

This chapter evaluates the Vienna Game AI Library from three key perspectives, namely applicability, performance and usability. The outcome is examined thoroughly in relation to the research questions stated in the first chapter of this thesis. Additionally, a discussion on limitations and improvements will be conducted as a final part of the chapter.

5.1. Applicability Evaluation

The applicability testing aims to demonstrate how the proposed library can be used in the development of a 2D RTS game. It serves as an extension to the examples discussed in the previous chapter as it offers a more complex scenario to showcase the library's capabilities. The chosen scenario is loosely based on *Age of Empires II* [Ensemble Studios, 1999], a well-known real-time strategy game, and aims to showcase several AI-driven agents that could be implemented in RTS games with the help of the library.

Civilian units are the characters that responsible for gathering food and resources, which in turn can be used to upgrade the town. Depending on their role, they are required to perform different actions, such as harvesting crops, chopping wood, or fishing. By using the collected materials, these units can then construct different buildings that are essential in training new units, advancing military strength, or trading between cities.

This kind of actions can be managed by FSMs, with the different potential activities representing states, such as “gather”, “mine”, or “hunt” when collecting resources, or “build” when constructing buildings. Changing between said states would be managed by in-game variables or events that would act as transitions. For instance, a farmer would change from “idle” to “gather” when the crops are fully done. Similarly, builders will enter the “build” state when enough construction resources are available, and exit it when the building is done. The “arrive” behaviour can be used by any unit when reaching its destination, in addition to the “face” technique to ensure that each unit is oriented towards its target.

Combat or military units ensure the protection of their towns on both land and sea, as well as destroy enemy military buildings and bases. Decision trees can be used to control which actions they may conduct based on a set of given conditions. For example, when encountering enemy units, these combat agents have the option to fight them, run away from them, or call for reinforcements and retreat until they arrive. The conditions to be checked may be related to the units' health or based on whether the military units are outnumbered by the enemy. The units can have different roles, such as defending the town by patrolling around it, or attacking enemy bases. In the early stage of the game they might start on the defence side, but once there are enough patrolling units that can look out for the town, some troops can be deployed to enemy bases to attempt conquering them.

5.1. Applicability Evaluation

All these actions would require decision trees to ensure a proper decision-making flow, and they may also be combined with state machines to enhance the process. Furthermore, the military units may take advantage of steering behaviours. “Pursue” can be used to follow retreating enemies, while “evade” is suitable when attempting to run away from potential threats. “Face” is also relevant, as the units need to be looking at the direction they are moving in. Moreover, the units can utilize the flocking algorithm to ensure smooth formation movement, either when patrolling around the town or when navigating from one town to another.

Priests are special units that are trained in temples and can heal wounded units, as well as convert the religion of enemy units. They are especially relevant in situations where the player does not intend to conquer an enemy town through military force, but instead it attempts to convert it, as the newly converted units come under the player’s control. Such units would require decision-making systems to determine when to heal a wounded unit. If state machines are used, the priest may have two states, such as “idle” and “heal”. A wounded unit entering the healing range of the priest would act as a transition that would activate the “heal” state. Decision trees can also be used to control this flow. Additionally, both systems would be useful when deciding which enemy units need to be converted first.

Each of the stated units needs pathfinding to move efficiently within the game environment. Builders might need to move from one building to another, farmers might need to navigate between their houses and the farm, while military units and priests could move between towns. In each case, it is essential that the units are as efficient as possible while avoiding the obstacles within the game environment.

Animals are neutral units depicting food sources. They can come in various forms, namely livestock, wild, and fish. Livestock are animals such as cows, sheep, and goats, while wild animals can be deer, boars, and moose. Both can use the “wander” technique to mimic real-life animal movement within the game world, as they do not have specific goals to follow. Similarly to the other units, they can also benefit from the “face” technique when moving. When civilian units approach them, they might get scared and use the “evade” behaviour to run away from them, making the hunting process more challenging.

Having covered the different types of AI-driven agents, a specific game scenario can be explored. Similarly to *Age of Empires II*, the player starts the game by getting assigned a town with several units. The game world consists of multiple cities, the others being fully managed by AI in this case. During the game, the player can expand its city by gathering resources, constructing buildings, expanding its military forces, and sending priests to convert other cities. In order to offer a challenging and immersive game experience, the enemy cities will interact with the player through different campaigns. They might attack the player’s base or attempt to convert it, to which the player will need to respond in order to protect its territories. The goal of the game is to conquer all cities, either through military campaigns or by converting the religion of their population. The game can also vary in difficulty, which can be achieved by designing the AI agents with different levels of intelligence. For instance, in easy modes, AI-driven units might use simpler tactics, such as the “flee” behaviour instead of “evade”, making it easier for the pursuers to catch them.

5. Evaluation and Discussion

Also, their decision-making processes can be simplified by making military units fight even in cases when they are outnumbered without waiting for reinforcements. For higher difficulty levels, the AI-controlled characters can be more strategic, making decisions based on the actions of the player or the game environment, thus ensuring challenging and dynamic opponents.

This scenario focuses on providing intelligent opponents for the player by incorporating most of the techniques offered by the library. However, the units managed by the player might also benefit from some of these techniques, proving that they are not only applicable to AI-controlled characters. For example, units moving across the game map might use the pathfinding feature, allowing for efficient path calculation and obstacle avoidance. Additionally, several steering behaviours might be suitable depending on the unit types. For instance, the “face” behaviour can be used such that the moving units orientate themselves correctly towards their target, thus enhancing the realism of the characters. Moreover, the “arrive” behaviour is especially relevant for scenarios in which the player-controlled units need to stop smoothly when reaching their destination, preventing abrupt stops. Lastly, for groups of units, the flocking algorithm can be used to ensure cohesive movement while maintaining an average velocity and avoiding crowding.

This section examined a scenario of a complex real-time strategy game in which the different AI techniques provided by the Vienna Game AI Library can be combined to create an immersive gameplay. The case study proves that the library contains enough functionality to develop a 2D RTS game from an AI perspective, thus answering the first research question stated in the beginning of this paper.

5.2. Performance Evaluation

The performance testing was conducted on a Lenovo Legion Pro 7 laptop with a 13th Gen Intel(R) Core(TM) i9-13900HX, 2.20 GHz processor, 32GB RAM (Random-access memory), 24 cores, running a 64-bit version of Windows 11 Pro. Although many profiling tools were initially considered for this evaluation, it was ultimately decided to not use any, but instead to opt for a custom and more flexible approach. Depending on its performance requirements, each feature is tested in one of two ways, namely execution time and memory usage. Structure padding is not considered when discussing memory usage. For several features, new demos were created in order to conduct such testing, being derived from the demos created to showcase the library’s features and slightly refactored to align with the specific evaluation requirements.

Regarding pathfinding, execution time is the most critical aspect that needs to be evaluated as the feature’s efficiency can highly influence the performance of the application. To calculate the time complexity of the pathfinding technique, it is essential to analyse every part of the A* algorithm. Let N denote the total number of NavMesh nodes. Initialising the *parents* array that stores the nodes forming the path takes $O(N)$ time. The main loop of the algorithm runs while the *openSet* priority queue is not empty, which could also take $O(N)$ if all nodes are added to the set, this depicting the worst case scenario. The next relevant step is processing the neighbours of each considered node.

Since the processed number of neighbours of a node can vary from 2 to 8 (depending on the heuristic function used and the position of the node within the NavMesh), let k denote the average number of neighbouring nodes. Thus, the inner loop that processes the neighbours takes $O(k)$ per node. With these factors considered, it can be stated that, so far, the time complexity is $O(N \cdot k)$, which can be referred to as $O(N^2)$ if k nears N . However, this would only occur if $N \leq 8$, so for $N > 8$, the time complexity is simply $O(N)$. Inserting and deleting nodes from the priority queue is also relevant, as both operations take $O(\log(N))$ time individually. It is assumed that, during the algorithm, the size of the priority queue is also N , considering the worst case scenario in which all NavMesh nodes are added to the set. Therefore, the overall time complexity for the A* algorithm is $O(N \cdot \log(N))$. In terms of space complexity, the algorithm will require $O(N)$ space as the *parents* array and *openSet* priority queue might store all nodes. The execution times of pathfinding using only the A* algorithm on different NavMesh sizes is provided later in this section for a better comparison to its improved version.

One critical factor that can affect the time complexity of the A* algorithm is the heuristic function, since it can either reduce or increase the number of nodes explored during the algorithm. The two functions provided by the library, the Manhattan and Euclidean distances, are considered to be suitable heuristics for grid-based navigation meshes. Neither will overestimate the cost between a node and the target node, thus resulting in finding the most optimal path. Therefore, these functions do not affect the previously calculated time complexity.

The geometric preprocessing step is also critical for performance due to the multitude of computations it is in charge of. This process iterates through all the NavMesh nodes, therefore it requires $O(N)$ time. During each iteration, every region is examined, thus adding a complexity of $O(R)$, with R denoting the number of regions. For every region, each of its assigned nodes are looped through to calculate the path to the nodes from the other regions. This results in a $O(n_R)$ complexity, with n_R denoting the average number of nodes per region, along with a $O(A)$ complexity which is the time it takes to find the path using the A* algorithm. This step was calculated above, hence $O(A) = O(N \cdot \log(N))$. By combining all these factors, it can be stated that the time complexity for this process is $O(N^2 \cdot \log(N) \cdot R \cdot n_R)$. Regarding space complexity, the most relevant structure is the list that stores all preprocessed paths from each node to each region. In the worst case scenario, it can take up to $O(N \cdot R \cdot n_P)$ space, n_P denoting the average number of nodes found on each path, meaning it would store the maximum number of paths for all node-region pairs. By enabling multithreading, the time complexity will decrease, while the space complexity will increase. Delegating the computation to the threads will ensure less execution time, but it will require more space as each thread gets a copy of the NavMesh nodes to avoid data racing. This might cause a performance drop but only during the memory allocation phase.

It is essential to discuss the *NavMesh* class as well, since it is responsible for enabling the aforementioned pathfinding techniques, in addition to storing relevant data. Initialising an instance of the *NavMesh* class takes $O(x \cdot y)$ time, where x is the number of nodes on the X-axis, and y denotes the node count on the Y-axis.

5. Evaluation and Discussion

In terms of memory usage, an object of this class requires at least 17 bytes. 1 byte is used to indicate whether preprocessing was performed, 8 bytes are for storing the width and height of the mesh, and 8 bytes hold the pointer to the list of regions. The NavMesh nodes are stored into an array of *NodeData* elements, thus taking $O(N)$ space, with N being the total number of nodes. A *NodeData* object occupies 24 bytes to hold node-related data. Inside the *NavMesh* class, there is also an array that stores the neighbours of each node, using $O(N \cdot k)$ memory, with k as the average number of neighbours.

A performance test was conducted on two versions of the pathfinding feature provided by the library. The execution times taken by these techniques were measured on different NavMesh sizes, with the amount of nodes ranging from 500 to 9500. By examining these execution times, it is expected to determine how impactful the improvements made to the technique are compared to the standard version. As stated previously, the speed of the preprocessing step depends heavily on the chosen number of threads. The amount of required threads can be set arbitrarily, depending on the CPU cores available on the user computer. Increased parallelism leads to faster processing, thus 24 threads were used to perform the evaluation, this being the most suitable amount for the device used.

A custom *Timer* class was created to measure the execution time of any given function through the use of the *chrono* C++ header. It provides two methods to handle counting, namely *start()* and *end()*. To retrieve the time taken between the two function calls, the class provides a method called *getDuration()*. This returns the total elapsed time in microseconds (μs) in order to get a clear overview of any function’s computational time. The reason why microseconds are used instead of milliseconds is that the features have execution times too short to be easily represented using milliseconds.

Number of NavMesh nodes	Execution times		Speed-up
	Standard version	Improved version	
500	54.63 μs	17.11 μs	3.19
1500	177.13 μs	12.91 μs	13.72
2500	206.43 μs	9.86 μs	20.94
3500	278.26 μs	11.79 μs	23.60
4500	767.26 μs	24.91 μs	30.80
5500	627.91 μs	15.78 μs	39.79
6500	779.82 μs	21.89 μs	35.62
7500	1232.17 μs	29.14 μs	42.28
8500	1712.86 μs	33.69 μs	50.84
9500	2047.55 μs	25.94 μs	78.93

Table 5.1.: Standard vs. Improved Pathfinding: Execution Times

The two versions were performed 100 times on the different sizes of navigation meshes and an average execution time was calculated. It is important to note that all measurements are done in release mode, to ensure optimal performance and remove any debug-related overhead. Table 5.1 depicts the outcome of this test, displaying the time required to find a path between two nodes with and without geometric preprocessing.

The start and end nodes were chosen consistently for each evaluation, being located at the top-left and bottom-right corners of the navigation mesh respectively. This ensured that the path chosen to be calculated had the maximum length such that the process doesn't result in too little calculations. The geometric preprocessing step used the 10x10 size for the regions regardless of the NavMesh size. The maximum number of nodes that was tested was 9500 due to the increasing preprocessing time, as it takes approximately 40 minutes to compute the paths for this number. Retrieving a preprocessed path takes little time, but it is essential to emphasise that it might still include a call to the A* algorithm within the target region. Consequently, the size of each region is important for performance, with smaller sizes resulting in a higher number of regions, which increases the number of loops conducted per node.

The speed-up column illustrates how fast the improved version is, in contrast to the standard version. It was calculated by dividing the average time of the standard version to the time of the improved one, rounded to the first two decimal points. As expected, it is evident that the improvements made to the pathfinding feature lead to better performance at runtime, with the speed-up increasing as the number of nodes grows. One aspect that will need to be monitored or further improved is the time the preprocessing takes for the heavy calculations, especially when the navigation mesh consists of a high number of nodes. Another observation is that there is one case in which pathfinding with geometric preprocessing can take longer than the standard version. There are situations where a path cannot be computed within the target region, which can occur if there are obstacles within the region that were not accounted for during the preprocessing step. This could cause issues for the game, and one way to solve it is always to check whether the process can retrieve a valid path. If none can be found, the A* algorithm can be called directly within the region using a nearby node as the starting point.

Decision trees need to be evaluated differently due to the abstract way in which the data structure is constructed. Their time and space complexities depend heavily on the implementation of each decision node since they are represented by generic classes and their implementation differs depending on the developer's requirements.

As illustrated in the previous chapter, the initialisation of a node is done by first creating a custom class which can vary in size based on the chosen variables. Similarly, the main method of a decision node, *makeDecision()*, is a pure virtual method whose complexity depends on its custom implementation. Adding child nodes to an existing node also differs in terms of time and space complexity, as the method takes a generic class as argument. The operations of retrieving a child node based on a given index and getting the number of total child nodes require $O(1)$ time and space. These are direct operations that do not rely on the size of the arrays that store the children of decision nodes. However, the array used to keep track of a node's children takes $O(c_S \cdot c_N)$ space, with c_S being the average space a child node takes, and c_N the number of children the respective node has. Deleting a decision node involves deleting each individual child node, thus requiring $O(c_T \cdot c_N)$ time, with c_T denoting the average time required to delete a child node. Creating the root of the tree requires the creation of a decision node, which, as stated previously, depends heavily on the custom class implementation.

5. Evaluation and Discussion

Resetting the tree implies deleting each node and its children and with d being the depth of the tree, this operation would require $O((c_T \cdot c_N)^d)$ time. Regarding the `update()` method that traverses the tree, the time complexity is $O(d)$. At each level of the tree, at most one decision is made when calling the `makeDecision()` function, thus enabling a node on the next tree level to continue the decision-making process.

While these calculations do not provide much information when discussing the decision tree from an abstract perspective, it might be helpful to look at a few examples from the demo offered by the library. Instead of complexity, the memory usage will be examined instead. The scenario discussed in the previous chapter involves two decision trees, one for each non-player character. Although constructed as a binary tree, the decision tree of the burglar is the most complex in the given example. Figure 5.1 depicts the structure of the said tree. The diagram was created with the help of the Lucid software [79].

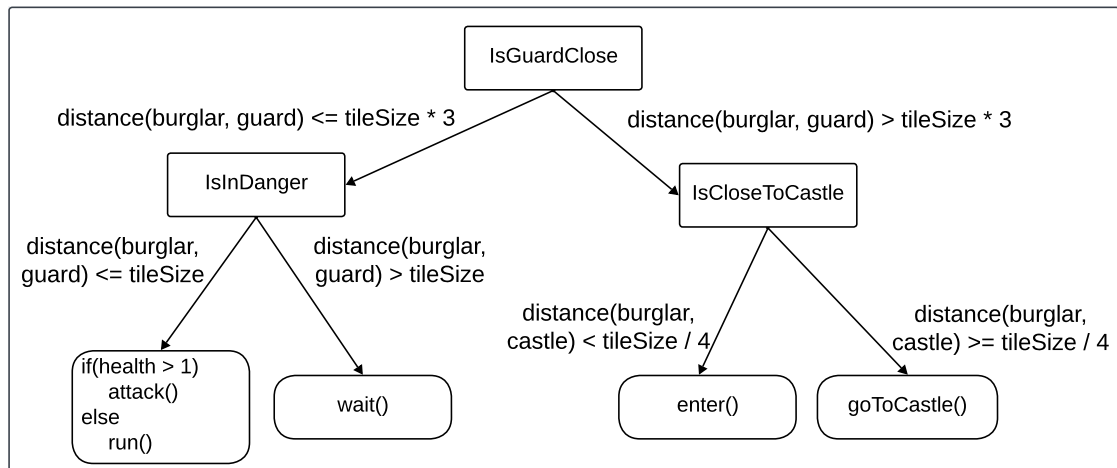


Figure 5.1.: Decision Trees Demo: Burglar’s Decision Tree

In this example, the rectangles represent decision nodes and the edges between them depict the conditions that would enable them. The shapes with rounded corners illustrate the logic of the two child nodes. The `IsGuardClose` depicts the root tree that can delegate actions to its child nodes, `IsInDanger` and `IsCloseToCastle`. The `makeDecision()` method of these two nodes contains logic that directly controls the actions of the character. With the help of the `sizeof` operator, the sizes in bytes of each stated class were retrieved. The root has a size of 48 bytes, while its child nodes, read from left to right, have 56 and 48 bytes respectively. Each class mostly consists of integer and float values to depict game variables, thus being relatively simplistic as they do not involve heavy computations.

Even though precise memory usage cannot be determined, it can be assumed that the generic classes used to represent decision nodes will have a minimum size of 8 bytes due to the use of the `makeDecision()` abstract method. By using the method, a virtual table is used to manage it at runtime, and a virtual pointer is responsible to point to the said table, which occupies 8 bytes. Such measurements can be considered when implementing decision trees, since their abstract form limits the ability to evaluate their complexity.

State machines are evaluated similarly, as their performance also depends on their custom implementation which is defined by the developer. *Transition* objects do not require significant time or space during initialisation or deletion due to their simple structure. Any instance of this class stores a target state and a callback function that checks whether said state should be activated. Both of these are saved as pointers, thus they each occupy 8 bytes, however the callback function might require more space depending on its custom implementation. The constructor of the class gets the target state and the condition as arguments, then assigns them to the respective internal member variables. Therefore, creating a transition takes $O(1)$ time. The destructor is explicitly declared as a default one, thus not implying any complexity considerations.

States are slightly more complex as they store more information. Each instance has three callback functions that are called depending on the status of the state, namely *onEnterCallback()*, *onUpdateCallback()*, and *onExitCallback()*. Similarly to the case of transitions, each of these functions has a memory allocation of at least 8 bytes, in addition to the cost of their custom implementation. Each state also stores an array of its outgoing transitions which are responsible for activating other states based on given criteria. As the array consists of *Transition* classes, it can be stated that the allocated memory is $O(t_S \cdot t_N)$, where t_S is the average space taken by an instance of this class, and t_N is the average number of transitions a state may have. Adding a transition to said array typically should take $O(1)$ time if the array has enough storage. However, if its capacity has been reached and more memory needs to be allocated, the entire array will resize by copying its elements to a larger block of memory. The adding operation will then take $O(t_A)$ time, with t_A denoting the number of transitions in the array. Retrieving the array of transitions has a time complexity of $O(1)$.

The state machine that manages all states and transitions stores an array of all its states, along with a pointer to the current active state which occupies 8 bytes. The space complexity of this class is dependent on the storage allocated for each individual state, and it can be referred to as $O(s_S \cdot s_N)$, where s_S is the average space required by a *State* instance, and s_N is the number of stored states. Creating a state takes $O(1)$ time with its memory usage starting from 8 bytes used for the pointer to the new *State* instance, in addition to the cost of its custom implementation. Retrieving the target state has a $O(1)$ time complexity. The *update()* method that updates the process loops through the transitions of the current active state until one of their callbacks is triggered. In that case, a new state is set as the current active state and the appropriate functions are called to exit the previous active one, as well as to enter the new one. With t_N as the average number of transitions of the current state, the time complexity of this function is $O(t_N)$ in addition to the cost of each individual invoked callback.

The steering behaviours will be discussed individually, with the exception of those that are nearly identical, differing in little lines of code. Before that, it is important to discuss the *Boid* class, which enables the use of such techniques. An instance of this class stores an unsigned integer depicting its identifier within a group of boids, four floats depicting its minimum and maximum speed, its last saved rotation, the angle used in the “wander” behaviour, and two *Vec2f* instances that represent its position and velocity.

5. Evaluation and Discussion

Together, they require 52 bytes. Creating a *Boid* object takes $O(1)$ time and space as it assigns said data to the values given as arguments to the constructor. The functions in this class are typically setters and getters, which do not involve high time and space complexity. The methods that are relevant for performance evaluation are the one that provide the functionality for steering. As discussed in the previous chapter, once each function responsible for enabling steering calculates its respective steering force, the *applySteeringForce()* method needs to be invoked such that the force is added to the velocity of the boid. Lastly, the *updatePosition()* function is called such that the position of the boid is updated. These two functions, as well as the steering techniques, take $O(1)$ time and space per boid.

To determine the performance of each individual steering technique, separate tests were conducted on applications running at 60 FPS. This is a standard benchmark that ensures that the features can run smoothly on devices with lower specifications. As one frame takes 1/60 seconds (approximately 16670 microseconds), the goal is to measure the frame time consumed by the calculations during each algorithm. The reason behind choosing this type of evaluation is to determine whether the computations performed for each steering behaviour might take too much frame time, especially since, in games, there are other critical processes that must be handled within the same frame, such as rendering, game logic, and physics simulations. Such processes must not be interfered with as the game performance will be negatively impacted.

The individual demos created for each behaviour were used in this evaluation, along with the *Timer* class to monitor the execution time. In the examples for “seek”, “flee”, “pursue”, and “evade”, an input-controlled agent moves using the arrow keys. The respective steering behaviour is applied to several entities, ranging from 10000 to 100000. Each evaluation was conducted over a span of 1000 frames, measuring the time required to calculate the steering force in each frame, from which an average execution time was computed. As in the previous test cases, the tests were performed in release mode.

One aspect that was observed during testing is that the execution times are slightly lower when the target does not move, however the difference is very small. A reason why this could occur is that the target position might be cached if not changed, thus dynamic targets would require more processing. To determine the highest execution times for the “seek”, “flee”, “pursue”, and “evade” behaviours, the input-controlled agent is constantly moving during their respective evaluations.

The results from applying this test on the “seek” and “flee” behaviours can be seen in Table 5.2. They are discussed together as the operations performed by each technique are computationally identical. The average time taken to calculate either of the two steering forces per boid is on average approximately 0.01 μs . Up to 50000 entities, the application ran smoothly and no decrease in performance was visible. However, after beyond this point, the rate of FPS lowered to a maximum of 58, with the most noticeable frame lag observed at 100000 entities. When evaluating the program with a FPS rate of 144, the common standard for high-refresh-rate displays, the performance drop was much more noticeable, averaging at 70 FPS. This may be attributed to the rendering processes since the number of entities displayed on the screen is significantly high.

5.2. Performance Evaluation

Number of Entities	“Seek” Behaviour	“Flee” Behaviour
10000	316.85 μs	303.01 μs
25000	751.89 μs	727.57 μs
50000	954.02 μs	938.11 μs
75000	1,022.07 μs	1,192.11 μs
100000	1,348.11 μs	1,208.39 μs

Table 5.2.: “Seek” and “Flee” Behaviours: Execution Times

The same evaluation was performed on the “pursue” and “evade” behaviours, with its results being showcased in Table 5.3. As expected, since the techniques involve more operations than the regular “seek” and “flee” behaviours, they require slightly more execution time. Just as in the previous example, the performance decrease started when using around 50000 entities. The FPS rate did not lower below 58, however the highest tested number of entities caused the application to run less smoothly.

Number of Entities	“Pursue” Behaviour	“Evade” Behaviour
10000	473.75 μs	455.84 μs
25000	1,057.51 μs	974.29 μs
50000	1,341.82 μs	1,253.01 μs
75000	1,379.24 μs	1,489.54 μs
100000	1,713.92 μs	1,747.04 μs

Table 5.3.: “Pursue” and “Evade” Behaviours: Execution Times

The demo for the “arrive” behaviour consists only of the entities that attempt to reach a target determined by a mouse click on the application screen. The same factors are considered for evaluation as in the previous test cases, with the only difference being that the target will not move throughout the process, in order to include the time it takes for the entities to fully stop moving. The results are depicted in Table 5.4. Since the target was static in this testing, this technique’s average execution time is smaller than that of the “seek” behaviour, although it implies slightly more calculations. Similarly to the previous test cases, the performance decrease becomes visible as the number of entities increases beyond 50000.

Number of Entities	“Arrive” Behaviour
10000	243.12 μs
25000	464.34 μs
50000	750.88 μs
75000	785.62 μs
100000	883.75 μs

Table 5.4.: “Arrive” Behaviour: Execution Times

5. Evaluation and Discussion

Based on the evaluation performed on the four steering behaviours, it can be assumed that the time taken to compute the respective steering forces accounts for a maximum of approximately 11% of a frame time. The “seek”, “flee”, and “arrive” behaviours require less time, this upper limit referring to the other two mentioned techniques. While the steering behaviours scale well in this evaluation, it is important to note that there are other factors that can affect performance, which were not included in these scenarios, such as additional entities requiring more calculations, more complex game logic, and rendering background elements.

Number of Entities	“Wander” Behaviour	Average FPS
10000	10,221.50 μs	59
25000	25,477.31 μs	35
50000	52,079.29 μs	20
75000	78,526.20 μs	12
100000	102,543.01 μs	9

Table 5.5.: “Wander” Behaviour: Execution Times

The “wander” steering behaviour involves more calculations, thus it is the most time-consuming out of the discussed features. This is due to the additional randomness, which generates different directions each frame, leading to frequent vector recalculations. Moreover, using the sine and cosine functions involves higher computational cost compared to the simple arithmetic operations applied in the previous cases. Table 5.5 illustrates the outcome of the evaluation, with a new column depicting the average FPS rate recorded for each test case. Although not noticeable, the frame rate dropped to 59 when 10000 entities were used. The frame lag became visible beyond this threshold, ranging from 9 to 35 depending on the entity count. As 10000 entities require more than half the frame time, an upper limit for these elements can be defined. The steering behaviour should be used on a maximum of approximately 8000 entities at once in an application, in order to not interfere with other complex processes. This recommendation refers to this specific case, where no improvements are considered and there are no other factors that can affect performance. The limit can be adjusted depending on each game’s requirements and priorities. Later in this chapter, several optimisations are discussed which could speed up the calculations for this technique and potentially increase said upper limit.

Number of Entities	“Face” Behaviour	Average FPS
10000	457.48 μs	60
25000	981.91 μs	60
50000	2,206.82 μs	35
75000	3,100.79 μs	23
100000	3,807.69 μs	15

Table 5.6.: “Face” Behaviour: Execution Times

In the same example refactored for testing the “wander” behaviour, an evaluation of the “face” behaviour was also performed. The function only returns the amount of degrees to which a boid should rotate to face its direction of movement and does not apply any force on it. As specified previously, it is used when rendering *Boid* instances. The evaluation focuses on measuring the time required to perform the calculations, and its outcome is presented in Table 5.6. The average FPS rate is displayed as well, as it is relevant to examine it with a higher entity count.

The calculations for this steering behaviour are also time consuming, although they require much less time in comparison with the “wander” technique. With a noticeable performance decrease when using 50000 entities, it can be stated that the maximum optimal entity count for this behaviour is between 25000 and 35000, as tests revealed a lower FPS rate for numbers above this threshold. One reason why this technique requires much more time than some of the others discussed may be the use of the *std::atan2* trigonometric function which has a higher computational cost than the arithmetic operations used for the previous techniques. This function returns the arc tangent of two numbers expressed in radians, which is later converted to degrees.

Regarding the group movement, the three individual steering techniques that compose the flocking algorithm will be first examined in terms of time and space complexity. Their execution times will be measured within the context of the algorithm, as separately they would lead to higher combined times since each technique iterates through the array of boids independently.

In the flocking algorithm, all three behaviours are handled within a single loop. The “separation” behaviour requires $O(b)$ time, with b denoting the number of boids within a flock. The function occupies 12 bytes of memory, 8 being allocated for the steering force to be calculated, and 4 for temporarily storing the distance between the current boid and every other member of the flock. The “alignment” behaviour is also $O(b)$ in time complexity, but has a higher memory usage, namely 16 bytes, as it also stores the number of neighbours of the current boid. Lastly, the “cohesion” behaviour requires the same time and space as the “alignment” technique.

The flocking algorithm is enabled by the *update()* function from the *Flock* class. It involves looping through the flock and calling the *doFlocking()* method for each individual boid. This part alone is $O(b)$ in time complexity, with b being the number of boids. Inside the *doFlocking()* function, there is an additional loop which handles the three aforementioned steering behaviours.

Flock Size	The Flocking Algorithm	Average FPS
1000	3,467.53 μs	60
2500	8,225.24 μs	60
5000	29,838.51 μs	35
7500	70,720.70 μs	17
10000	123,925.01 μs	9

Table 5.7.: The Flocking Algorithm: Execution Times

5. Evaluation and Discussion

Considering every other operation within this function takes $O(1)$ time, the flocking algorithm performs with a total time complexity of $O(b^2)$ as for each boid an additional iteration through the flock is performed. Regarding space, the algorithm requires 36 bytes per boid to store the three steering forces, the distance to the other flock members, the number of neighbours, and its speed. To evaluate its execution times, the same approach as in the previous test cases was followed. The algorithm ran on different flock sizes for 1000 frames, then the average of its execution times was computed, as depicted in Table 5.7.

In this evaluation, the entity count was chosen much smaller compared to the other steering behaviours. This was caused by the significant decrease in FPS rate when using more than 5000 entities. Up to approximately 3000 boids, the algorithm does not involve any frame lag and the application runs smoothly. Beyond this threshold, performance is notably affected. It is safe to assume that anything above 2500 boids will impact the application as the average execution time for this entity count is in the vicinity of half a frame time. Without any optimisations considered and depending on the game, this may negatively impact performance.

5.3. Usability Evaluation

To test the library's usability, a survey was conducted, which was completed by 20 people. Its questions targeted three areas, namely participants' background, game AI usage, and the Vienna Game AI Library. The background information section consists of three questions whose purpose is to gain insight into the professional experience of each participant. The questions cover the work industry, years of programming experience, and the programming languages and/or game engines they work with primarily. The second section aims to assess the experience or knowledge the participants may have in regards to game AI technologies. Its questions revolve around game AI frameworks and libraries and check whether the survey takers have worked on or contributed to any open source or in-house tools. The section also includes questions on resources the participants might have relied on while using said frameworks, as well as potential AI technologies they adopted in the development of non-player character behaviour. The final area comprises in-depth questions about the proposed library, focusing on the individual techniques offered by the library, as well as general preferences that participants might have. The content of the survey can be found in Appendix A.1.

The answers of the first section indicate that the industry in which all participants work is computer science, but their specialisations differ considerably. The respondents' years of experience range from 2 to 13, with the majority reporting 6. In terms of programming languages, the majority of participants chose C#/.NET as the language they work with primarily, followed by C++ as the second most popular option. Regarding game engines, the participants mentioned Unity, Unreal, and Godot, the most chosen one being the Unity Engine. This section aimed to understand the background of the participants, as their experience and knowledge is relevant for the remaining survey questions.

The second section, game AI usage, revealed that less than half of the respondents had previously worked on game AI frameworks and libraries. None of them, however, had contributed to one. The participants who reported experience with AI frameworks or libraries stated several resources they rely on when using said tools, with most mentioned being documentation, forums, and tutorials. A minority of respondents had worked on developing complex non-player character behaviour, specifically behaviour trees, pathfinding with NavMesh, PID (Proportional-integral-derivative) controllers, state machines, and neural networks. The purpose of this section was to comprehend the extent of each participant's experience with game AI technologies. While many responses indicate that there is a lack of such experience and knowledge, some of them provided information that consolidated several decisions made for the proposed library. For instance, it is clear that the majority of participants rely on documentation and tutorials when using a library. The library provides both resources, which are available in the README file of the GitHub repository [68], as well as in the files generated by Doxygen.

The third and final question started by introducing the idea of an external AI library in the context of developing AI-driven characters for strategy games. When asked which programming language the participants would prefer the library to be in, the majority of people were divided into two equal groups, one stating C++/C, and the other C#. The next popular choice was Python, followed by JAVA and JavaScript. This question illustrated a preference for C++/C, confirming that the choice of the programming language for the library was well-justified. The participants were also asked how they would prefer including the library in their project and were given three options, namely "source code", "prebuilt binaries", and "I don't mind". The majority opted for having the library in source code form, which is how the Vienna Game AI Library is intended to be provided. Additionally, the participants were questioned which features and algorithms they would expect to be supported by the library in the given scenario. The most mentioned ones were pathfinding, state machines, decision-making processes (with a focus on behaviour trees), and formation control. Other suggestions were given, such as steering behaviours, collision avoidance, resource management, and reinforcement learning. The answers to this question solidified the rationale behind the feature choices of the library, although instead of behaviour trees, decision trees are provided.

Having covered the general requirements for a game AI library, the next questions focused on the features provided by Vienna Game AI Library aiming to determine whether it is easy to understand and use. Therefore, the participants were first asked to estimate the lines of code they would need to implement their own pathfinding algorithm.

The answers varied considerably, but the majority stated that they would require less than 100 lines. The participants were then given a code snippet from the library and were asked to estimate how many lines they think they would need to implement the feature by using the library, including the creation of the navigation mesh. The number of estimated lines decreased, with the majority stating they would require less than 50 lines. The conclusion that can be drawn from these answers is that the participants expect to write less code for pathfinding if they use the Vienna Game AI Library, but they overestimate the lines they would need, potentially due to their unfamiliarity with the library.

5. Evaluation and Discussion

The survey also included a question related to the preprocessing step, to check whether they find it useful. All of them stated that they would use such a feature, especially if the game environment is static and the obstacles and other elements within it do not change too often. They believe that it could improve overall performance and memory if the NavMesh is not dynamic, which is the current case for the library. As mentioned earlier, the preprocessing step is most useful for situations in which the NavMesh has little to no changes, such that there is no overload of computations done before runtime.

The same approach was followed for decision trees and state machines. However, it is hard to estimate the lines of code needed to implement either feature, as it heavily depends on the amount of individual elements created for each feature. Even so, the answers reveal that the participants believe they would need to write much less code using the Vienna Game AI Library than if they developed the features on their own. For decision trees, the majority of respondents estimated less than 75 lines for their own implementation, and less than 20 if using the library. Regarding state machines, the estimations also decreased significantly, with most participants expecting around 100 lines for their own approach, and less than 50 with the help of the library.

Similar questions were asked for each individual steering behaviour, along with the flocking algorithm. By showing the participants the methods that would allow them to use these techniques, it was expected to determine whether a clear understanding of the steering implementation was formed. While it is difficult to come a conclusion based on the answers as they vary considerably, a pattern was observed. It seems that for each individual behaviour, the majority of participants believe they would require less lines of codes when using the library. For the flocking algorithm, the majority of responders believed that they would need between 50 and 500 lines of code for their own implementation, but less than 50 lines when using the library.

While it is evident that participants would expect to write less code when using the library, these questions aimed to demonstrate that the functionality of each feature is easily understandable. It is believed that this outcome has been achieved as in most cases the participants anticipated the appropriate amount of code.

The final two questions aimed to assess the opinion of the participants with regards to the Vienna Game AI Library. When asked if they would use it to develop strategy games based on the given code snippets, half of them said “likely”, one chose “very likely”, and the remaining opted for “neutral”. The remaining two options were “unlikely” and “very unlikely”, but neither was chosen. The last question revealed their opinion on the following statement: “Vienna Game AI Library is a comprehensive, easy to use and practical library for developing strategy games.”, with options varying from “strongly agree” to “strongly disagree”. The majority of respondents agreed with the said statement, several stated that they strongly agree with it, and the remaining opted for neither agreeing nor disagreeing.

As an optional question, the participants were asked to give any feedback they may have on the library. One respondent said that the API is straight forward and easy to understand, but a concern would be the potential need for custom functionality and extensibility of the library. However, this concern is addressed as the library is delivered in source code form and is adequately documented, which would allow for such capabilities.

Another participant also mentioned that the library is easy to use and can save time for developers, while making a suggestion in regards to development, namely multi-agent agent collaboration and consensus. The last feedback received stated that it is hard to make a judgement on the library based only on the given information, but declared that the pathfinding feature especially seems powerful. The results of the survey are available in Appendix A.2.

The conclusions drawn so far were made based on observing the results and identifying certain tendencies. In order to justify and strengthen these conclusions, the Wilcoxon rank-sum test was performed. Also known as the Wilcoxon–Mann–Whitney test, it is a statistical test that compares two groups of data that come from the same participants in order to determine whether they differ significantly [95]. Two CSV (Comma-separated values) files were created, one containing the estimated lines of code for the participants’ own implementations, and the other including their estimates when using the library. Let X represent random values chosen from the “OwnImplementationDataset” file which contains the estimations of own implementations, while Y depicts random values chosen from the other dataset. The test evaluates the hypothesis that the probability of X being greater than Y is equal to the probability of Y being greater than X . If true, this hypothesis will reveal that the distributions of the two datasets are identical.

Otherwise, the alternative hypothesis would be chosen, meaning that the distributions differ from one other. The test was conducted in order to demonstrate that the estimations are lower when using the Vienna Game AI Library, hence it attempts to prove the alternative hypothesis. To perform this evaluation, the R project [96] was used, which provides the R programming language and several statistical techniques that are utilized in hypothesis testing. Listing 5.1 illustrates the code written in the said language, which loads the two stated files and prints the results of the Wilcoxon rank-sum test.

```

1 own_dataset<-read.csv("/path/to/OwnImplementationDataset.csv")
2 vgain_dataset<-read.csv("/path/to/VGAILImplementationDataset.csv")
3 own <- own_dataset[,2:20]
4 vgain <- vgain_dataset[,2:20]
5 do<-function() {
6   for(l in 1:10) {
7     own_res<-unlist(own[l,!is.na(own[l,])], use.names=FALSE)
8     vgain_res<-unlist(vgain[l,!is.na(vgain[l,])], use.names=FALSE)
9     print(own_dataset[l,1])
10    print(wilcox.test(own_res, vgain_res, alternative="greater",
11                    exact=FALSE))
12  }
13 do()

```

Listing 5.1: Wilcoxon Rank-Sum Test: Code Snippet

The main outcome of this test can be seen in Table 5.8, with the complete results being provided in Appendix A.3. The first column illustrates each individual question, while the second depicts p -value, which determines which hypothesis is true. If it is greater than 0.05, the original hypothesis is correct and the two datasets do not differ significantly. Otherwise, the alternative hypothesis is proven.

5. Evaluation and Discussion

Question	P-value
How many lines of code do you think you would need to implement your own path finding algorithm?	0.00305
How many lines of code do you think you would need to implement your own decision trees?	0.03741
How many lines of code do you think you would need to implement your own state machines?	0.07865
How many lines of code do you think you would need to implement the seek steering technique?	0.06418
How many lines of code do you think you would need to implement the flee steering technique?	0.07087
How many lines of code do you think you would need to implement the pursue steering technique?	0.05424
How many lines of code do you think you would need to implement the evade steering technique?	0.03771
How many lines of code do you think you would need to implement the arrive steering technique?	0.02766
How many lines of code do you think you would need to implement the wander steering technique?	0.07336
How many lines of code do you think you would need to implement the flocking steering technique?	0.05621

Table 5.8.: Wilcoxon Rank-Sum Test: Results

The results demonstrate that for some questions, the answers between the two groups are statistically considerably different, some are at the limit, while others are similar. However, *p-value* is remarkably low for each individual question, thus it is reasonable to assume that the differentiating factor, namely “using the Vienna Game AI Library”, had a significant influence on the answers.

5.4. Discussion

Having evaluated the library from the three different perspectives, it is essential to discuss the results in relation to the research questions stated in the beginning of this paper, along with limitations and further potential improvements.

The applicability testing provided a case study of a standard real-time strategy game inspired by the well-known *Age of Empires II* game. The different features offered by the proposed library were proven to be useful when managing AI-driven units, as they would facilitate navigation, steering, and decision-making processes. The examples showcasing each feature, with the exception of pathfinding, also depicted small scenarios that can be found in RTS games. For instance, the demo for the finite state machine presented a regular worker unit that gathers materials and drops them off at a given location.

For decision trees, a basic guard-burglar interaction was illustrated in the context of castle defence. Lastly, the steering behaviours example showed a group of animals showcasing life at a farm. Therefore, it is reasonable to conclude that the small examples and the applicability evaluation successfully answered the first research question: “Does the proposed library provide all features that are required to develop 2D real-time strategy games from an AI perspective?”. Additionally, it is plausible to assume that the functionality of the library can cover the overall strategy genre, even though the focus was initially set on the real-time subgenre. In RTS games, AI-driven agents must make decisions and calculations in real-time, thus the library’s ability to manage computationally heavy processes is critical. Turn-based games do not require such immediate responses from the AI techniques. Consequently, the library is suitable for both subgenres.

During the performance evaluation, time and space complexities were examined, along with measuring execution times and memory usage. By analysing these factors, several conclusions could be drawn in regards to optimisations and limitations. Pathfinding was shown to be efficient at runtime when using geometric preprocessing and enabling multithreading. However, as previously stated, increasing the size of the navigation mesh implies heavy computational time. For static navigation meshes, it is worth performing this step, then store it in a file in order to allow for quick retrieval of the preprocessed paths, rather than recomputing the data each time when needed.

For dynamic meshes, this process might be less useful as it implies recalculating the paths whenever there is a change in the NavMesh’s structure. For this specific case, the two alternative features that were considered in the planning stage, also mentioned in Chapter 2, might be more suitable options. The ALT algorithm [75] involves preprocessing as well, but the amount of pre-calculations is much smaller than in the version used in the proposed library. Given several “landmark” nodes, the algorithm first computes the distance from each said type of node to every other regular node in the mesh. It still uses the A* algorithm to find the path between two nodes, but the heuristic function is different. With a start and a target node given, the function creates an imaginary triangle between said nodes and a landmark node. As the preprocessing part had been performed, the distances between the landmark node and the other two nodes are easily retrieved, with these distances depicting two sides of the imaginary triangle. The third side has to be at least as long as the difference between the other two sides, so it is computed for every pre-calculated landmark. The longest “at least” value is then the one that will be used. While this method might save time during the preprocessing step, it involves more computation at runtime which may lead to a higher execution time. The ALP algorithm [76] is another alternative, that introduces a new heuristic based on polygons, instead of triangles. Depending on the game requirements, either option might prove itself to be more appropriate.

There is also the possibility to create different game world representations, other than navigation meshes, such as tile graphs, Voronoi diagrams, or waypoint systems. The A* algorithm can also be improved to run more efficiently. Chapter 2 introduced several variations of this algorithm, including HPA* and IDA* for static pathfinding, and LPA*, RTAA*, and AD* for dynamic pathfinding.

5. Evaluation and Discussion

Having discussed potential improvements, it is possible to examine factors that limit this feature. Throughout the evaluation process, the process of finding the most optimal path ran smoothly at runtime regardless of the NavMesh size. With the highest tested amount of nodes, namely 9500, the A* algorithm alone consumed approximately 12% of the frame time to retrieve the path, percentage that can be considered minimal. Enabling geometric preprocessing with multithreading, the execution time is significantly lower, being around 79 times faster. Evidently, this option involves high processing time before the application runs, representing the primary drawback of the feature. Neither version significantly impacts the frame rate when targetting 60 FPS, however high-refresh-rate displays might encounter issues when using only the A* algorithm as it would consume 30% of the frame time for the highest node count. Considering other processes that need to be handled during games, such as rendering, physics simulation, or input management, it is possible to determine an upper limit for the NavMesh size depending on which version of pathfinding is used. For the standard version using only the A* algorithm, the risk of the computational overhead for 60 FPS applications is low for any node count below 9500. With geometric preprocessing, in order for the pre-calculation step to take a reasonable amount of time, the appropriate number is between 5000 and 5500 nodes.

In regards to the decision trees and state machines, time and space complexities were evaluated, rather than execution times and memory usage. This is due to their abstract nature, as it is hard to determine their performance without their custom implementation. Regardless, there are several improvements that can be discussed in relation to their limitations.

Decision trees are the most efficient when the tree is balanced [16]. This aspect needs to be taken into consideration when implementing this feature, especially in situations where the tree grows large and the nodes involve substantial computations. While not related to performance, an option to improve the feature is making it less predictable. Including random elements in the tree will ensure variation, which is essential in games such that players do not lose interest. While decision trees are easy to understand and use, they are not very suitable in dynamic environments due to their rigid structure. On the other hand, behaviour trees have a modular and hierarchical structure that allows for flexibility and scalability, in addition to efficiency as their underlying systems involve fewer tree traversals.

Regarding optimisations for finite state machines, a well known and used option is represented by the hierarchical state machines. They provide more flexibility and scalability than the regular FSMs by grouping states into levels, along with reducing redundancy. This feature is most suitable for complex systems that involve a multitude of state and transitions.

In terms of steering, each individual technique performed relatively well considering the 60 FPS target, apart from the “wander” and “face” behaviours. As examined previously, their computational overhead is due to the trigonometric operations that are essential in calculating their respective forces. One optimisation that can be applied to all steering techniques is adding the *-ffast-math* compiler option as a flag to the CMake file or build command, since it would enable faster execution of such mathematical operations.

This, however, might lead to less accurate results due to the use of floating-point numbers. In games, this is typically not a major concern as it is not essential to have precise calculations, with approximations often offering satisfactory results. Another improvement is using custom approximation methods instead of the `std::atan2()`, `std::sin()`, and `std::cos()` functions, which would be more efficient, although not as precise.

The flocking algorithm can also benefit from using the `-ffast-math` flag for a better performance. A conclusion derived from its respective evaluation is that, in an application targeting 60 FPS, the algorithm cannot use more than 2000 boids at once while preserving the frame rate. Several aspects of the algorithm has potential for improvement, as demonstrated by Li et al. [97], which targets cohesion, and Lee et al. [98] that aims for overall efficiency.

The performance evaluation determined the best use cases for all features, thus the library’s practicality in game development currently depends on the requirements of each specific game. The library provides support for handling non-player characters in real-time strategy games, particularly on a smaller scale. It effectively manages navigation, decision-making processes, and steering, being an appropriate tool for games with limited number of non-player characters. While it performs well in such scenarios, its performance depends heavily on the number of characters created for the game, as well as external factors such the complexity of the game logic and other computationally heavy processes.

The usability testing was conducted to determine whether developers can understand the intent behind the Vienna Game AI Library. Based on the answers to each question, along with the received feedback, it was concluded that the survey participants grasped the given API features, suggesting that the library is user-friendly. It is plausible to state that the library is also easy to use, as the participants seemed to recognize that they would need minimal lines of codes to enable the pathfinding, steering, and flocking when using the library. The decision trees and state machines were more challenging to estimate due to their abstract structure. Even so, without considering the creation of their respective structures, enabling each feature requires at maximum two lines of code. Overestimations still occurred, however this could be attributed to the fact that not enough API information was given, as one participant also suggested. According to the individual feedback received at the end of the survey, it was determined that there are other features that developers may expect from the library, each of which is discussed in the last chapter of this thesis. Nonetheless, with the given set of techniques the library supports, and as per the outcome of the survey, the second research question (“Are the features provided by the proposed library easy to use?”) was successfully answered.

The third research question, namely “Is the proposed library well documented and demonstrated?”, can be answered by examining source code comments, as well as the files generated by Doxygen, along with the README file from the GitHub repository [68] and the examples showcasing each feature. Each class and function included in the library has source comments describing its purpose, its arguments, as well as the data it returns or operates on. Such descriptions are available also for variables, whether they are global or part of classes and functions. This information is used by Doxygen to generate a detailed HTML file that provides insight into the library’s structure and content.

5. *Evaluation and Discussion*

Regarding demonstrations, the examples discussed in the previous chapter serve as a guide for developers, as they include every aspect of their respective feature. Additionally, the README file contains instructions on how the entire project can be built and started, along with explanations of the library's features in the context of the demo examples. Given the extensive range of resources, it is reasonable to assume that the library is well documented and demonstrated.

The last research question was "Is the proposed library easy to integrate into C++ projects?". This question can be answered successfully only when considering the Windows operating system, as the library was created and tested only on such a system. The README file offers a list of all prerequisites that are needed in order to run the GitHub project. Moreover, two scripts are provided, one for building the library and another one for running it. If developers want to use it outside of this environment, the only requirement is to add the header file to their projects. Therefore, integrating the library into C++ projects is straightforward.

This chapter illustrated three ways in which the Vienna Game AI Library was evaluated. The outcome of each individual testing was examined in relation to the research questions stated in the beginning of this paper, but also considering the initial intent behind the library. The next chapter will conclude the work presented in this thesis, and discuss future plans for the library.

6. Conclusion and Future Work

The proposal of this thesis is the Vienna Game AI Library, a C++ single-header file that encompasses a collection of AI techniques to be used in the development of 2D real-time strategy games. The main factor of motivation is the lack of public frameworks that contain enough functionality to aid game developers create such games. Additionally, there is a need for a game AI library in the Vienna suite of projects used in courses at the Faculty of Computer Science of the University of Vienna.

The paper began by discussing academic papers written on this topic, as well as addressing published software that has similar content with the Vienna Game AI Library. The respective chapter reinforced the core motivation, but also offered valuable insight on potential features the library may benefit from. The following chapter introduced key concepts regarding games and their usage of AI. It provided background information such that the technical aspects of the library can be comprehended easily.

The third chapter focused solely on the Vienna Game AI Library, discussing its setup, content, structure, and implementation. Information on the chosen tools for building, documentation, and deployment was given, as well as the rationale behind each decision. Then the list of techniques supported by the library was presented, in addition to UML diagrams that introduced underlying connections between all feature components. The implementation phase was thoroughly discussed, including each feature and its respective demo. Pathfinding was examined in regards to its main aspects, namely navigation meshes and the A* algorithm, along with its improvements given by enabling geometric preprocessing and multithreading. Regarding the decision-making flow, decision trees and finite state machines were individually examined. Lastly, steering behaviours were explained in detail, for both individual and formation units. For all mentioned features code snippets were provided to showcase their internal implementation in the library, along with how they can be used externally.

The evaluation chapter focused on answering the research questions stated in the first chapter. Through the applicability testing, it was proven that the library contains enough functionality to develop 2D real-time strategy games from an AI perspective. Moreover, considering all demo examples, fully implemented or simply discussed, that showcased the potential of the library, it is reasonable to state that the Vienna Game AI Library can cover the overall strategy genre, not just the real-time strategy subgenre. The usability testing demonstrated that the library is user-friendly and easy to understand and use. The performance testing determined limitations for each feature, evaluating time and space complexities, execution times, and memory usage. The remaining two research questions were answered separately as they targeted documentation and integration within other C++ projects. A discussion on the outcome of these tests was conducted, where improvements and limitations were highlighted.

6. Conclusion and Future Work

Consequently, building upon the extensive analysis in these chapters, it can be stated that the Vienna Game AI Library proved through a series of evaluations that it achieved the purpose of providing a collection of AI features that can aid developers in making 2D real-time strategy games.

As stated throughout the thesis, the library will remain within the Faculty of Computer Science of the University of Vienna, and it is intended to be further worked on during future theses. Since it will be expanded by other algorithms, it is relevant to outline several suggestions that were brought up during the usability testing. The survey conducted for this evaluation gathered opinions from the different participants who were asked what they would expect from a game AI library.

One of the features mentioned is represented by behaviour trees. Their benefits were covered in previous chapters, however it is important to mention that they would bring additional flexibility and scalability and are worth considering, depending on the future requirements of the library.

Collision avoidance was another suggestion, which led to the consideration of other steering behaviours that were not included in the original plans of the library. There is a multitude of techniques that can enrich steering, such as collision, obstacle and wall avoidance, path following, and leader following.

Resource management is especially relevant in strategy games as it would allow the AI agents to make decisions depending on their resources. This would ensure intelligent opponents for the player, thus keeping the gameplay engaging.

Lastly, the library may expand beyond non-player character management. The field of machine learning can also be considered, as reinforcement learning and neural networks were suggested in the survey. They can be used for optimisation purposes, game balancing, or player behaviour prediction. Procedural content creation is an option be regarded as well since it would allow for creating dynamic and challenging game levels or scenarios for the player.

Bibliography

- [1] K. Karpouzis and G. Tsatiris. *AI in (and for) Games*. Springer International Publishing, 01 2022. https://doi.org/10.1007/978-3-030-76794-5_3.
- [2] J. Jiang, Y. Kuang, and H. Shen. The Application of AI for the Non Player Character in Computer Games. In *2011 International Conference on Computational and Information Sciences*, pages 1049–1050. IEEE, 2011. <https://doi.org/10.1109/ICCIS.2011.318>.
- [3] R. Santamaria and Contributors. Raylib, 2023. <https://www.raylib.com/>. Last accessed on: 30/06/2024.
- [4] P. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Path. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. <https://doi.org/10.1109/tssc.1968.300136>.
- [5] X. Cui and H. Shi. A*-based Pathfinding in Modern Computer Games. *International Journal of Computer Science and Network Security (IJCSNS)*, 11(1):125–130, 2011. <https://api.semanticscholar.org/CorpusID:6458879>.
- [6] A. Primanita, R. Effendi, and W. Hidayat. Comparison of A* and Iterative Deepening A* algorithms for non-player character in Role Playing Game. In *2017 International Conference on Electrical Engineering and Computer Science (ICECOS)*, pages 202–205, 2017. <https://doi.org/10.1109/ICECOS.2017.8167134>.
- [7] S. Koenig, M. Likhachev, and D. Furcy. Lifelong Planning A*. *Artificial Intelligence*, 155(1):93–146, 2004. <https://doi.org/10.1016/j.artint.2003.12.001>.
- [8] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun. Anytime Dynamic A*: An Anytime, Replanning Algorithm. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling*, volume 5, pages 262–271, 2005. <https://dl.acm.org/doi/10.5555/3037062.3037096>.
- [9] S. Koenig and M. Likhachev. Real-time adaptive A*. In *Proceedings of the International Conference on Autonomous Agents*, pages 281–288, 05 2006. <http://dx.doi.org/10.1145/1160633.1160682>.
- [10] A. Bleiweiss. GPU Accelerated Pathfinding. In *Graphics Hardware*, pages 65–74, 01 2008. <https://doi.org/10.2312/EGGH/EGGH08/065-074>.

Bibliography

- [11] C. W. Reynolds. Steering Behaviors For Autonomous Characters. In *Proceedings of the Game Developers Conference*, pages 763–782, 1999. <https://api.semanticscholar.org/CorpusID:12156361>.
- [12] Steering Behaviors For Autonomous Characters | by Craig Reynolds. <https://www.red3d.com/cwr/papers/1999/gdc99steer.html>. Last accessed on 16/09/2024.
- [13] C. W. Reynolds. Flocks, Herds, and Schools: A Distributed Behavioral Model. *SIGGRAPH '87 Conference Proceedings*, 21(4):25–34, 1987. <https://doi.org/10.1145/37402.37406>.
- [14] Boids | Background and Update | by Craig Reynolds. <https://www.red3d.com/cwr/boids/>. Last accessed on 16/09/2024.
- [15] H. Fathy, O. A. Raouf, and H. Abdelkader. Flocking Behaviour of Group Movement in Real Strategy Games. In *2014 9th International Conference on Informatics and Systems*, 2014. <https://doi.org/10.1109/INFOS.2014.7036679>.
- [16] I. Millington and J. Funge. *Artificial Intelligence for Games, Second Edition*. Morgan Kaufmann Publishers Inc., 2nd edition, 2009. <https://dl.acm.org/doi/10.5555/1795711>.
- [17] C. Kingsford and S. L. Salzberg. What are decision trees? *Nature biotechnology*, 26(9):1011–1013, 2008. <https://doi.org/10.1038/nbt0908-1011>.
- [18] D. Thomas and A. Hunt. State Machines. *IEEE Software*, 19(6):10–12, 2002. <https://doi.org/10.1109/MS.2002.1049380>.
- [19] R. Alur, S. Kannan, and M. Yannakakis. Communicating Hierarchical State Machines. In *Automata, Languages and Programming*, volume 1644, pages 169–178. Springer Berlin Heidelberg, 05 1999. https://doi.org/10.1007/3-540-48523-6_14.
- [20] V. Sklyarov. Hierarchical Finite-State Machines and Their Use for Digital Control. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(2):222–228, 1999. <https://doi.org/10.1109/92.766749>.
- [21] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. [https://doi.org/10.1016/0167-6423\(87\)90035-9](https://doi.org/10.1016/0167-6423(87)90035-9).
- [22] M. Colledanchise and P. Ögren. *Behavior Trees in Robotics and AI: An Introduction (1st ed.)*. CRC Press, 1st edition, 2018. <https://doi.org/10.1201/9780429489105>.
- [23] S. Ontañón. Informed Monte Carlo Tree Search for Real-Time Strategy games. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8, 2016. <https://doi.org/10.1109/CIG.2016.7860394>.
- [24] S. L. Tomlinson. The long and short of steering in computer games. *International Journal of Simulation Systems, Science and Technology*, 5:1–2, 06 2004.

- [25] M. Kapadia and N. I. Badler. Navigation and steering for autonomous virtual humans. *WIREs Cognitive Science*, 4(3):263–272, 2013. <https://doi.org/10.1002/wcs.1223>.
- [26] H. Danielsiek, R. Stüer, A. Thom, N. Beume, B. Naujoks, and M. Preuss. Intelligent Moving of Groups in Real-Time Strategy Games. In *2008 IEEE Symposium On Computational Intelligence and Games*, pages 71–78, 2008. <https://doi.org/10.1109/CIG.2008.5035623>.
- [27] M. Naveed, D. Kitchin, and A. Crampton. Monte-Carlo Planning for Pathfinding in Real-Time Strategy Games. In *Proceedings of PlanSIG 2010. 28th Workshop of the UK Special Interest Group on Planning and Scheduling joint meeting with the 4th Italian Workshop on Planning and Scheduling*, pages 125–132, 2010. <http://eprints.hud.ac.uk/9242/>.
- [28] N.A. Mas’udi, E.M.A Jonemaro, M.A. Akbar, and T. Afrianto. Development of Non-Player Character for 3D Kart Racing Game Using Decision Tree. *Fountain of Informatics Journal*, 6(2):51–60, 2021. <https://doi.org/10.21111/fij.v6i2.4678>.
- [29] Y. Miyake, Y. Shirakami, K. Shimokawa, K. Namiki, T. Komatsu, J. Tatsuhiro, P. Prasertvithyakarn, and T. Yokoyama. A Character Decision-Making System for FINAL FANTASY XV by Combining Behavior Trees and State Machines. In *Game AI Pro 3*, pages 145–157, 2017. <http://dx.doi.org/10.4324/9781315151700-11>.
- [30] J. Lin, J. He, and N. Zhang. Application of behavior tree in AI design of MOBA games. In *2019 IEEE 2nd International Conference on Knowledge Innovation and Invention (ICKII)*, pages 323–326. IEEE, 2019. <https://doi.org/10.1109/ICKII46306.2019.9042660>.
- [31] S. Balapriya and N. Srinivasan. Patrolling AI Systems in Video Games. *International Journal of Recent Technology and Engineering (IJRTE)*, 8:2479–2485, 11 2019. <http://www.doi.org/10.35940/ijrte.D6957.118419>.
- [32] A. Shaout, B. King, and L. Reisner. Real-Time Game Design of Pac-Man Using Fuzzy Logic. *International Arab Journal of Information Technology*, 3(4):315–325, 01 2006.
- [33] R. Fronek, B. Göbl, and H Hlavacs. Procedural Creation of Behavior Trees for NPCs. In *Entertainment Computing - ICEC 2020*, pages 285–296. Springer International Publishing, 2020. https://doi.org/10.1007/978-3-030-65736-9_26.
- [34] R. Dworzanski and H. Hlavacs. Shaping AI Behavior: A Q-Learning Driven Approach to Automatic Behavior Tree Creation. In *2024 IEEE International Conference on Artificial Intelligence and eXtended and Virtual Reality (AIxVR)*, pages 241–245. IEEE Computer Society, 01 2024. <https://doi.ieeecomputersociety.org/10.1109/AIxVR59861.2024.00039>.

Bibliography

- [35] P. Schwab and H. Hlavacs. PALAIS: A 3D Simulation Environment for Artificial Intelligence in Games. In *6th International Symposium on AI & Games*, 04 2015. <https://api.semanticscholar.org/CorpusID:14171223>.
- [36] M. Lent, P. Carpenter, R. McAlinden, and P. G. Tan. A Tactical and Strategic AI Interface for Real-Time Strategy Games. *AAAI Conference on Artificial Intelligence*, 01 2004. <https://api.semanticscholar.org/CorpusID:1067595s>.
- [37] R. Rosalina, A. Sengkey, G. Sahuri, and R. Mandala. Generating intelligent agent behaviors in multi-agent game AI using deep reinforcement learning algorithm. *International Journal of Advances in Applied Sciences*, 12:396, 12 2023. <http://dx.doi.org/10.11591/ijaas.v12.i4.pp396-404>.
- [38] I. Sagredo-Olivenza, P.P. Gómez-Martín, M.A Gómez-Martín, and P.A. González-Calero. Combining Neural Networks for Controlling Non-player Characters in Games. In *Advances in Computational Intelligence*, pages 694–705. Springer International Publishing, 2017. https://doi.org/10.1007/978-3-319-59147-6_5.
- [39] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv*, 06 2016. <https://doi.org/10.48550/arXiv.1606.01540>.
- [40] Yuka | A Javascript library for developing Game AI. <https://mugen87.github.io/yuka/>. Last accessed on: 01/05/2024.
- [41].gdx-ai. <https://github.com/libgdx/gdx-ai>. Last accessed on: 12/08/2024.
- [42] libGDX - libGDX. <https://libgdx.com/>. Last accessed on: 12/08/2024.
- [43] BrainAI. <https://github.com/ApmeM/BrainAI>. Last accessed on 01/05/2024.
- [44] Unity Movement AI. <https://github.com/sturdyspoon/unity-movement-ai>. Last accessed on 01/05/2024.
- [45] Pygame. <https://github.com/pygame/pygame>. Last accessed on 01/05/2024.
- [46] behaviac. <https://github.com/Tencent/behaviac>. Last accessed on 01/05/2024.
- [47] Recast Navigation. <https://github.com/recastnavigation/recastnavigation>. Last accessed on 11/09/2024.
- [48] OpenSteer: Steering Behaviors for Autonomous Characters. <https://github.com/meshula/OpenSteer>. Last accessed on 11/09/2024.
- [49] Godot Engine - Free and open source 2D and 3D game engine. <https://godotengine.org/>. Last accessed on: 01/05/2024.
- [50] Free Fuzzy Logic Library. <https://ffll.sourceforge.net/>. Last accessed on 16/09/2024.

- [51] S. Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., 2002. <https://dl.acm.org/doi/10.5555/515547>.
- [52] GameAI. <https://github.com/in-op/GameAI>. Last accessed on 01/05/2024.
- [53] The Unity Machine Learning Agents Toolkit (ML-Agents). <https://github.com/Unity-Technologies/ml-agents>. Last accessed on 12/08/2024.
- [54] I. Granic, A. Lobel, and R. Engels. The Benefits of Playing Video Games. *American Psychologist*, 69(1):66–78, 2014. <https://doi.org/10.1037/a0034857>.
- [55] J. Gregory. *Game Engine Architecture (1st ed.)*. A K Peters/CRC Press, 2009. <https://doi.org/10.1201/9781315267845>.
- [56] Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. <https://unity.com/>. Last accessed on: 01/05/2024.
- [57] The most powerful real-time 3D creation tool - Unreal Engine. <https://www.unrealengine.com/en-US>. Last accessed on: 01/05/2024.
- [58] Vulkan | Cross platform 3D Graphics. <https://www.vulkan.org/>. Last accessed on 27/08/2024.
- [59] OpenGL - The Industry Standard for High Performance Graphics. <https://www.opengl.org/>. Last accessed on 27/08/2024.
- [60] Metal Overview - Apple Developer. <https://developer.apple.com/metal/>. Last accessed on 27/08/2024.
- [61] The Vienna Vulkan Engine (VVE). <https://github.com/hlavacs/ViennaVulkanEngine>. Last accessed on: 27/08/2024.
- [62] The Vienna Physics Engine (VPE). <https://github.com/hlavacs/ViennaPhysicsEngine>. Last accessed on: 27/08/2024.
- [63] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013. <https://dl.acm.org/doi/10.5555/2543987>.
- [64] A. Williams. *C++ Concurrency in Action*. Manning, 2nd edition, 2019. <https://ieeexplore.ieee.org/document/10280302>.
- [65] R. Graham. An Introduction to Utility Theory. In *Game AI Pro*, pages 67–80, 09 2019. <https://doi.org/10.1201/9780429055058-6>.
- [66] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, 1965. [https://doi.org/10.1016/S0019-9958\(65\)90241-X](https://doi.org/10.1016/S0019-9958(65)90241-X).
- [67] J. Orkin. Applying Goal-Oriented Action Planning to Games. In *AI Game Programming Wisdom, Vol. 2*, pages 217–228, 2008. <https://api.semanticscholar.org/CorpusID:5989921>.

Bibliography

- [68] Vienna Game AI Library. <https://github.com/hlavacs/ViennaGameAIBLibrary>. Last accessed on 24/10/2024.
- [69] CMake - Upgrade Your Software Build System. <https://cmake.org/>. Last accessed on 15/02/2024.
- [70] Ninja, a small build system with a focus on speed. <https://ninja-build.org/>. Last accessed on 15/02/2024.
- [71] Clang C Language Family Frontend for LLVM. <https://clang.llvm.org/>. Last accessed on 15/02/2024.
- [72] Doxygen. <https://www.doxygen.nl/>. Last accessed on 15/02/2024.
- [73] A. Rafiq, T. A. A. Kadir, and S. N. Ihsan. Pathfinding Algorithms in Game Development. *IOP Conference Series: Materials Science and Engineering*, 769, 2020. <http://dx.doi.org/10.1088/1757-899X/769/1/012021>.
- [74] M. Zikky. Review of A* (A Star) Navigation Mesh Pathfinding as the Alternative of Artificial Intelligent for Ghosts Agent on the Pacman Game . *EMITTER International Journal of Engineering Technology*, 4(1):141–149, 2016. <https://doi.org/10.24003/emitter.v4i1.117>.
- [75] A. V. Goldberg and C. Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, 04 2005. <https://dl.acm.org/doi/10.5555/1070432.1070455>.
- [76] N. H. Campbell Jr. Computing Shortest Paths Using A*, Landmarks, and Polygon Inequalities (Abstract). *CoRR*, abs/1603.01607, 2016. <https://doi.org/10.48550/arXiv.1603.01607>.
- [77] J. Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Game Developers Conference 2006*, 2006. <https://api.semanticscholar.org/CorpusID:62493110>.
- [78] T. B. Yoon, K. H. Park, J. H. Lee, and K. M. Lee. User Adaptive Game Characters Using Decision Trees and FSMs. In *Agent and Multi-Agent Systems: Technologies and Applications*, pages 972–981. Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-72830-6_103.
- [79] Lucid | The Leading Visual Collaboration Platform. <https://lucid.co/>. Last accessed on 14/11/2024.
- [80] Understanding Steering Behaviors: Flee and Arrival. <https://code.tutsplus.com/understanding-steering-behaviors-flee-and-arrival--gamedev-1303t>. Last accessed on 03/12/2024.

- [81] Understanding Steering Behaviors: Pursuit and Evade. <https://code.tutsplus.com/understanding-steering-behaviors-pursuit-and-evade--gamedev-2946t>. Last accessed on 03/12/2024.
- [82] Understanding Steering Behaviors: Wander. <https://code.tutsplus.com/understanding-steering-behaviors-wander--gamedev-1624t>. Last accessed on 03/12/2024.
- [83] 5.5 Wander Steering Behavior - The Nature of Code. <https://www.youtube.com/watch?v=ujsR2vcJlLk>. Last accessed on 24/10/2024.
- [84] Boids algorithm - augmented for distributed consensus. https://vanhunteradams.com/Pico/Animal_Movement/Boids-algorithm.html. Last accessed on 24/10/2024.
- [85] A* search algorithm. https://en.wikipedia.org/wiki/A*_search_algorithm. Last accessed on 03/12/2024.
- [86] Platformer Pack Medieval · Kenney. <https://kenney.nl/assets/platformer-pack-medieval>. Last accessed on 01/12/2024.
- [87] Toon Characters 1 · Kenney. <https://kenney.nl/assets/toon-characters-1>. Last accessed on 01/12/2024.
- [88] Medieval RTS · Kenney. <https://kenney.nl/assets/medieval-rts>. Last accessed on 01/12/2024.
- [89] Shape Characters · Kenney. <https://kenney.nl/assets/shape-characters>. Last accessed on 01/12/2024.
- [90] Animal Pack Redux · Kenney. <https://kenney.nl/assets/animal-pack-redux>. Last accessed on 01/12/2024.
- [91] Tiny Town · Kenney. <https://kenney.nl/assets/tiny-town>. Last accessed on 01/12/2024.
- [92] Fish Pack · Kenney. <https://kenney.nl/assets/fish-pack>. Last accessed on 01/12/2024.
- [93] Signika - Google Fonts. https://fonts.google.com/specimen/Signika/about?preview.size=57&lang=en_Latn. Last accessed on 01/12/2024.
- [94] raylib [text] example - Input Box. https://www.raylib.com/examples/text/loader.html?name=text_input_box. Last accessed on 24/10/2024.
- [95] Wilcoxon rank-sum test. https://en.wikipedia.org/wiki/Mann-Whitney_U_test. Last accessed on 10/11/2024.
- [96] The R Project for Statistical Computing. <https://www.r-project.org/>. Last accessed on 10/11/2024.

Bibliography

- [97] C. Li, Y. Yang, T. Huang, and X. Chen. An improved flocking control algorithm to solve the effect of individual communication barriers on flocking cohesion in multi-agent systems. *Engineering Applications of Artificial Intelligence*, 137:109110, 2024. /url<https://doi.org/10.1016/j.engappai.2024.109110>.
- [98] J. M. Lee, S. H. Cho, and R. A. Calvo. A Fast Algorithm for Simulation of Flocking Behavior. In *2009 International IEEE Consumer Electronics Society's Games Innovations Conference*, pages 186–190, 2009. <https://doi.org/10.1109/ICEGIC.2009.5293611>.

Acronyms

- AD*** Anytime Dynamic A*. 5, 63
- AI** Artificial Intelligence. 1–11, 13–18, 20, 23, 31, 34–38, 40–42, 45–48, 58, 59, 62, 63, 67, 68
- ALP** A*, Landmarks and Polygons. 20, 63
- ALT** A*, Landmarks and Triangle. 20, 63
- API** Application Programming Interface. 10, 12, 60, 65
- CPU** Central Processing Unit. 13, 19, 20, 30, 50
- FPS** Frames Per Second. 6, 54–58, 64, 65
- FSM** Finite State Machine. 6, 30, 35, 46, 64
- GOAP** Goal-Oriented Action Planning. 9, 15
- GPU** Graphics Processing Unit. 2, 5, 12, 13
- HPA*** Hierarchical Pathfinding A*. 4, 63
- HSM** Hierarchical State Machine. 6, 36
- IDA*** Iterative Deepening A*. 4, 63
- LPA*** Lifelong Planning A*. 5, 63
- MCTS** Monte Carlo Tree Search. 6, 10
- MOBA** Multiplayer Online Battle Arena. 7
- NavMesh** Navigation Mesh. 4, 25, 26, 28, 30, 48–51, 59, 60, 63, 64
- NPC** Non-Player Character. 1, 5–7, 9, 10, 14–17, 19, 20, 32, 36, 38
- RTAA*** Real-Time Adaptive A*. 5, 63
- RTS** Real-Time Strategy. 1, 4, 5, 8, 9, 19, 30, 36, 41, 46, 48, 62, 63

Acronyms

UML Unified Modeling Language. xi, 21–23, 67

VR Virtual Reality. 12

A. Appendix

A.1. Survey Questions

Background information

1. What industry do you work in?

2. How many years of programming experience do you have?

3. What programming languages / game engines do you work with primarily?

Game AI usage

4. Have you previously worked with game AI frameworks or libraries? (Unreal Engine AI, Unity ML agents, etc.). If yes, specify which.

Yes

No

5. What resources do you rely on when using these libraries?

- Documentation
- Tutorials
- Articles
- Forums (e.g., Stack Overflow)
- Demos
- Others (please specify)

6. Have you ever worked on a game with complex Non-Player Character behavior? If yes, what type of AI technologies or algorithms did you use?

Yes

No

7. Have you ever contributed to a game AI library? (open source, or in-house). If so, which one?

- Yes

- No

VGAIL

Imagine you are developing a strategy game and you are using an external library for features that manage Non-Player Characters.

8. What programming language would you prefer this library to be in?

- C/C++
- C#
- JAVA
- JavaScript
- Python
- Other (please specify)

9. How would you like to have the library included in your project?

- Source code
- Prebuilt binaries
- I don't mind

10. What features would you expect from this library? Are there any specific algorithms that you would like to see supported?

Path finding

Path finding algorithms determine the most suitable and / or most optimal path for Non-Player Characters such that they can navigate through the game world. Such algorithms include Dijkstra's algorithm, the A algorithm, Breadth-First Search etc.*

11. How many lines of code do you think you would need to implement your own path finding algorithm?

12. In the Vienna Game AI Library, the call for finding a path is as follows:

```
std::vector<Vec2ui> findPath(Vec2ui start, Vec2ui target)
```

where *Vec2ui* is a custom implementation of a 2D vector and the path finding algorithm used is the A* algorithm.

How many lines do you think you would need to implement pathfinding with the help of this library? (including creating the navigation mesh)

13. In the Vienna Game AI Library, there is an option to enable geometric preprocessing with multithreading such that finding the most optimal path does not take too long at runtime.

Geometric preprocessing is responsible for dividing the navigation mesh into regions and calculating the shortest distance between each node of the navmesh and each region. These distances would be stored, then retrieved at runtime whenever needed. This saves time as the distances would not be calculated at runtime which could impact the performance depending on the size of the navigation mesh. To ensure this process is smooth and fast, it can be used with multithreading.

Do you think you would use such a feature? Why / why not?

Decision trees

Decision trees deal with controlling a Non-Player Character's actions based on given information. They are a more organized and formalized set of if-then-else rules, while resembling the tree data structure due to their nodes which each can contain child nodes.

14. How many lines of code do you think you would need to implement your own decision trees?

15. In the Vienna Game AI Library, decision trees have a root and decision nodes. To create a root, the following method would be called:

```
template <class T, typename... Args>  
DecisionNode& createRoot(Args&& ...args)
```

While to create a child node, the following method would be used:

```
template <class T, typename... Args>  
DecisionNode& addChild(Args&& ...args)
```

How many lines of code do you think you would need to implement decision trees by using this library?

State machines

State machines provide ways to model NPC action-taking by defining a set of states and transitions between them, where the states are specific actions that the characters can perform, and the transitions represent methods to switch between states depending on information from within the game world.

16. How many lines of code do you think you would need to implement your own state machines?

17. In the Vienna Game AI Library, creating a state is done by calling:

```
State* createState()
```

To create a transition, the following method would be called:

```
void addTransition(State* targetState, std::function<bool()> callback)
```

How many lines of code do you think you would need to implement state machines by using this library?

Steering behaviors

Steering behaviors are responsible for adding a layer of realism to games by allowing NPCs to simulate intelligent movements. Vienna Game AI Library provides a list of individual steering techniques:

- *Seek* - allows for realistic movement towards a given target.
- *Flee* - allows for realistic movement of "running away" from a given target.
- *Pursue* - allows for realistic movement of trying to "catch" a target by anticipating its movement.
- *Evade* - allows for realistic movement of trying to "outrun" a target by anticipating its movement
- *Arrive* – allows the character to slow down before it reaches its destination such that it can stop smoothly.
- *Wander* – allows the character to “wander” randomly.

18. How many lines of code do you think you would need to implement a steering technique?

Seek

Flee

Pursue

Evade

Arrive

Wander

19. In the Vienna Game Ai Library, the following method is called to use the “seek” steering behavior, where *Vec2f* is a custom implementation of 2D vectors and *f32* is a float:

```
Vec2f seek(Vec2f targetPosition, f32 maxAcceleration)
```

How many lines of code do you think you would need to implement the “seek” steering behavior by using this library?

20. In the Vienna Game Ai Library, the following method is called to use the “flee” steering behavior, where *Vec2f* is a custom implementation of 2D vectors and *f32* is a float:

```
Vec2f flee(Vec2f targetPosition, f32 maxAcceleration)
```

How many lines of code do you think you would need to implement the “flee” steering behavior by using this library?

21. In the Vienna Game Ai Library, the following method is called to use the “pursue” steering behavior, where *Boid* is a custom class for the character and *f32* is a float:

```
Vec2f pursue(const Boid* target, f32 maxAcceleration, f32 maxPrediction)
```

How many lines of code do you think you would need to implement the “pursue” steering behavior by using this library?

22. In the Vienna Game Ai Library, the following method is called to use the “evade” steering behavior, where *Boid* is a custom class for the character and *f32* is a float:

```
Vec2f evade(const Boid* target, f32 maxAcceleration, f32 maxPrediction)
```

How many lines of code do you think you would need to implement the “evade” steering behavior by using this library?

23. In the Vienna Game Ai Library, the following method is called to use the “arrive” steering behavior, where *Vec2f* is a custom implementation of 2D vectors and *f32* is a float:

```
Vec2f arrive(Vec2f targetPosition, f32 slowRadius, f32 maxAcceleration)
```

How many lines of code do you think you would need to implement the “arrive” steering behavior by using this library?

24. In the Vienna Game Ai Library, the following method is called to use the “wander” steering behavior, where *f32* is a float:

```
Vec2f wander(f32 circleDistance, f32 circleRadius, f32 displacementRange, f32 maxAcceleration)
```

How many lines of code do you think you would need to implement the “wander” steering behavior by using this library?

Flocking

Flocking behavior is a group steering behavior, as it is used to create a cohesive and natural looking movement for an entire group. It simulates the movement of a group similar to a flock of birds or army of units.

25. How many lines of code would you expect to write to use the flocking algorithm?

26. In the Vienna Game Ai Library, the following method is called to use the flocking behavior, where *Boid* is a custom class for boids of a flock and *f32* is a float:

```
void flocking(f32 deltaTime, f32 separationRange, f32 perceptionRange,  
f32 avoidFactor, f32 matchingFactor, f32 centeringFactor, const  
std::vector<Boid*>& flock)
```

How many lines of code do you think you would need to implement the flocking behavior by using this library? (including the creation of the flock)

Overall

27. Based on the API presented in the previous questions, how likely are you to use the Vienna Game AI Library when developing a Real-Time Strategy game?

- Very unlikely
- Unlikely
- Neutral
- Likely
- Very likely

28. Based on the API presented in the previous questions, what do you think about the following statement:

Vienna Game AI Library is a comprehensive, easy to use and practical library for developing RTS games.

- Strongly Disagree
- Disagree
- Neither Agree Nor Disagree
- Agree
- Strongly Agree

A.2. Survey Results

Total number of answers: 20

1. What industry do you work in?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
GENERAL IT	11	55%
XR	4	20%
FRONTEND DEVELOPMENT	1	5%
HEALTHCARE IT	1	5%
BANKING IT	1	5%
RENEWABLE ENERGY	1	5%
AI	1	5%

2. How many years of programming experience do you have?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
6 YEARS	5	25%
3 YEARS	4	20%
4 YEARS	3	15%
7 YEARS	3	15%
2 YEARS	2	10%
5 YEARS	1	5%
10 YEARS	1	5%
13 YEARS	1	5%

3. What programming languages / game engines do you work with primarily?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
C# / .NET	7	35%
C++	6	30%
JAVASCRIPT	5	25%
TYPESCRIPT	5	25%
PYTHON	5	25%
UNITY ENGINE	3	15%
JAVA	3	15%
UNREAL ENGINE	2	10%
RUST	1	5%
LIQUID	1	5%
VUE	1	5%
FRONTEND TECHNOLOGIES	1	5%

4. Have you previously worked with game AI frameworks or libraries? (*Unreal Engine AI, Unity ML agents, etc.*)

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
NO	12	60%
YES	8	40%

5. If you answered 'yes' to the previous question, please specify some game AI frameworks or libraries you worked with.

ANSWER	NUMBER OF ANSWERS	PERCENTAGE (FROM PEOPLE WHO ANSWERED 'YES')	PERCENTAGE (FROM ALL PARTICIPANTS)
UNITY ENGINE AI	7	87.5%	35%
UNREAL ENGINE AI	4	50%	20%
GODOT	2	25%	10%
CRYENGINE AI	1	12.5%	5%
VIENNA LIBRARY	1	12.5%	5%

- Unity Engine AI: Unity ML agents, Behavior designer
- Unreal Engine AI: Behavior trees

6. What resources do you rely on when using these libraries (if you answered 'yes') or any libraries in general?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
DOCUMENTATION	14	70%
FORUMS	14	70%
TUTORIALS	13	65%
ARTICLES	5	5%
DEMOS	5	5%

7. Have you ever worked on a game with complex Non-Player Character behavior?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
NO	18	90%
YES	2	10%

8. If you answered 'yes' to the previous question, what type of AI technologies or algorithms did you use?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE (FROM PEOPLE WHO ANSWERED 'YES')	PERCENTAGE (FROM ALL PARTICIPANTS)
BEHAVIOR TREES	2	100%	10%
NAVMESH PATHFINDING	1	50%	5%
PID CONTROLLERS	1	50%	5%
STATE MACHINES	1	50%	5%
NEURAL NETWORKS	1	50%	5%

9. Have you ever contributed to a game AI library? (open source, or in-house)

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
NO	20	100%

Imagine you are developing a Real-Time Strategy game and you are using an external library for features that manage Non-Player Characters.

10. What programming language would you prefer this library to be in?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
C++/C	8	40%
C#	8	40%
PYTHON	7	35%
JAVA	5	25%
JAVASCRIPT	5	25%
TYPESCRIPT	1	5%
RUST	1	5%

11. How would you like to have the library included in your project?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
SOURCE CODE	11	55%
I DON'T MIND	7	35%
PREBUILT BINARIES	2	10%

12. What features would you expect from this library? Are there any specific algorithms that you would like to see supported?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
PATHFINDING	4	20%
STATE MACHINES	3	15%
DECISION MAKING	3	15%
○ BEHAVIOR TREES	2	10%
FORMATION CONTROL	2	10%
STEERING BEHAVIORS	1	5%
GROUP MOVEMENT	1	5%
COLLISION AVOIDANCE	1	5%
RESOURCE MANAGEMENT	1	5%
NPC COMMUNICATION	1	5%
REINFORCEMENT LEARNING	1	5%
NEURAL NETWORKS	1	5%
HEARING SYSTEMS	1	5%
SENSING	1	5%
MULTI-AGENT CONSENSUS		5%
CUSTOM HEURISTICS AND REWARDS	1	5%
OBJECT DETECTION	1	5%
PREBUILT FUNCTIONS FOR CHARACTERS	1	5%
BLACKBOARDS	1	5%

13. How many lines of code do you think you would need to implement your own path finding algorithm?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
5	1	5%
6	1	5%
10	1	5%
20	1	5%
30	1	5%
10-30	1	5%
70-100	1	5%
100	2	10%
120	1	5%
100-200	1	5%
150-200	1	5%
250	1	5%
200-300	2	10%
400	1	5%
1000-2000	1	5%
1000+	1	5%
2000	1	5%
NO ANSWER	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
<= 100	9	45%
100-500	7	35%
500+	3	15%
NO ANSWER	1	5%

14. How many lines do you think you would need to implement path finding with the help of VGAIL? (including creating the navigation mesh)

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	
1	2	
2	1	
3	1	
2-3	1	
10	2	
5-20	1	
10-20	1	
20	1	
30	1	
50	1	
50-60	1	
100	1	
100+	1	
<150	1	
200	1	

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
10-50	11	55%
50-100	3	15%
100+	3	15%

15. In Vienna Game AI Library, there is an option to enable geometric preprocessing with multi-threading such that finding the most optimal path does not take too long at runtime. Do you think you would use such a feature? Why / why not?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
YES	20	100%

Concerns:

- “If the moving objects also have collision avoidance between each other, the scene includes a lot of moving objects and therefore the benefit of pre-computing the distance between all objects may be less noticeable”
- “I would avoid it if there are many dynamic objects which might change the NavMesh, as I would be unsure about the performance of re-preprocessing frequently.”
- “Yes if the game world is static and terrain/buildings do not change too often”
- Performance when running multiple NPCs
- Memory

16. How many lines of code do you think you would need to implement your own decision trees?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
5	1	5%
7	1	5%
20	2	10%
30-50	1	5%
50	1	5%
50-70	1	5%
70	1	5%
<75	1	5%
100	1	5%
200	1	5%
300-1000+	1	5%
500	1	5%
500-1000	1	5%
2000+	1	5%
5000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
< 75	9	45%
75-500	3	15%
500+	4	20%

17. How many lines of code do you think you would need to implement decision trees by using VGAIL?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
3	2	10%
4	1	5%
5	1	5%
10	2	10%
20	4	20%
30-50	1	5%
60-70	1	5%
<75	1	5%
100	1	5%
200	1	5%
100-200	1	5%
250	1	5%
1000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 20	10	50%
20-100	3	15%
100+	5	25%

18. How many lines of code do you think you would need to implement your own state machines?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
6	1	5%
10	1	5%
15	1	5%
20	1	5%
30	1	5%
50	2	10%
90-100	1	5%
100	2	10%
200-500	1	5%
300-500	1	5%
300	1	5%
400	1	5%
1000+	1	5%
4000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
< 100	10	50%
200-1000	4	20%
1000+	2	10%

19. How many lines of code do you think you would need to implement state machines by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
3	1	5%
4	1	5%
10	1	5%
20	3	15%
30	2	10%
45	1	5%
30-50	1	5%
50-100	1	5%
100	1	5%
150	1	5%
200	1	5%
300	1	5%
300+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	4	20%
< 50	10	50%
50-300	5	25%
300+	1	5%

20. How many lines of code do you think you would need to implement the seek steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
3	1	5%
5	2	10%
10	2	10%
20	1	5%
30	2	10%
35	1	5%
50	1	5%
55-60	1	5%
60	1	5%
100	1	5%
200	1	5%
300	1	5%
300-400	1	5%
1000+	1	5%
2000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 60	12	60%
60-500	4	20%
500+	2	10%

21. How many lines of code do you think you would need to implement the *seek* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	3	15%
5	2	10%
10	3	15%
20	3	15%
30-45	1	5%
45	1	5%
50	2	10%
150	1	5%
200	1	5%
200+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 20	8	40%
20-50	5	25%
50+	5	25%

22. How many lines of code do you think you would need to implement the *flee* steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	1	5%
3	1	5%
5	1	5%
10	1	5%
14	1	5%
20	1	5%
20-25	1	5%
25	1	5%
35	1	5%
50	3	15%
100	2	10%
200	1	5%
300	1	5%
800+	1	5%
3000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	7	35%
25-100	5	25%
100+	6	30%

23. How many lines of code do you think you would need to implement the *flee* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	4	20%
10	3	15%
14	1	5%
10-20	1	5%
20	3	15%
25	1	5%
50	2	10%
150	1	5%
200	1	5%
300+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	12	60%
25-100	3	15%
100+	3	15%

24. How many lines of code do you think you would need to implement the *pursue* steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
5	2	10%
6	1	5%
10	1	5%
15	1	5%
25	2	10%
30-45	1	5%
50	2	10%
60	1	5%
100	1	5%
150	1	5%
200	2	10%
350	1	5%
2300+	1	5%
3000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	5	25%
25-100	6	30%
100+	7	35%

25. How many lines of code do you think you would need to implement the *pursue* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	4	20%
10	3	15%
15	1	5%
15-20	1	5%
20	2	10%
30	1	5%
35	1	5%
40	1	5%
50	1	5%
200	1	5%
250	1	5%
500+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	11	55%
25-100	4	20%
100+	3	15%

26. How many lines of code do you think you would need to implement the *evade* steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
5	2	10%
6	1	5%
10	1	5%
15	1	5%
20-25	1	5%
25	1	5%
30	1	5%
40	1	5%
50	1	5%
60	1	5%
100	1	5%
200	2	10%
250	1	5%
300	1	5%
2000+	1	5%
3000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	6	30%
25-100	5	25%
100+	7	35%

27. How many lines of code do you think you would need to implement the *evade* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	4	20%
10	3	15%
10-15	1	5%
15	1	5%
20	3	15%
30	1	5%
50	1	5%
60	1	5%
100	1	5%
200	1	5%
500+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	12	60%
25-100	3	15%
100+	3	15%

28. How many lines of code do you think you would need to implement the *arrive* steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
4	1	5%
6	1	5%
10	1	5%
12	1	5%
15	1	5%
25	1	5%
30	1	5%
40	2	10%
40-45	1	5%
50	3	15%
100	1	5%
150	1	5%
300	1	5%
1500+	1	5%
3000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	5	25%
25-100	8	40%
100+	5	

29. How many lines of code do you think you would need to implement the *arrive* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	4	20%
5	1	5%
10	2	10%
12	1	5%
15	1	5%
20	3	15%
25	1	5%
40	1	5%
50	2	10%
200	1	5%
400+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	12	60%
25-100	4	20%
100+2	2	10%

30. How many lines of code do you think you would need to implement the *wander* steering technique?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
5	3	15%
8	1	5%
10	1	5%
15	1	5%
20	1	5%
25	1	5%
30	2	10%
35-40	1	5%
50	1	5%
100	2	10%
150	1	5%
300	1	5%
2000+	1	5%
4000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	7	35%
25-100	5	25%
100+	6	30%

31. How many lines of code do you think you would need to implement the *wander* steering behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
1	3	15%
2	1	5%
3	1	5%
5	2	10%
15-20	1	5%
20	3	15%
25	1	5%
30	2	10%
50	1	5%
200	1	5%
300	1	5%
500+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 25	11	55%
25-100	4	20%
100+	3	15%

32. How many lines of code do you think you would need to implement the *flocking* behavior?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
3	1	5%
5	1	5%
10	1	5%
15	1	5%
30	1	5%
40	1	5%
50	2	10%
50-70	1	5%
80	1	5%
100	1	5%
100-200	1	5%
200	2	10%
300	1	5%
400	1	5%
2500+	1	5%
3000	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 50	6	30%
50-500	10	50%
500+	2	10%

33. How many lines of code do you think you would need to implement the *flocking* behavior by using this library?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
5	2	10%
6	1	5%
10	2	10%
15	1	5%
25	1	5%
20-30	1	5%
30	1	5%
35-40	1	5%
40	1	5%
50	3	15%
80	1	5%
100	1	5%
200	1	5%
600+	1	5%

Summary

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
CANNOT ESTIMATE	2	10%
< 50	11	55%
50-100	4	20%
100+	3	15%

34. Based on the API presented in the previous questions, how likely are you to use the Vienna Game AI Library when developing a Real-Time Strategy game?

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
LIKELY	10	50%
NEUTRAL	9	45%
VERY LIKELY	1	5%

35. Based on the API presented in the previous questions, what do you think about the following statement:

Vienna Game AI Library is a comprehensive, easy to use and practical library for developing RTS games.

ANSWER	NUMBER OF ANSWERS	PERCENTAGE
AGREE	14	70%
STRONGLY AGREE	4	20%
NEITHER AGREE OR DISAGREE	2	10%

36. FEEDBACK

- “The API is fairly straight forward and easy to use and comprehend. The only downside could be that when developing any game there is a need for custom functionality and extensibility that this library might not have. If the source code is also provided and it is extensively documented, then it will allow developers to use it to it's full potential.”
- “The library seems easy to use and saves some time. I would really like to see implementation of multi-agent collaboration and consensus, which is one of the more difficult parts to implement and abstract into a comprehensive API.”
- “I definitely haven't seen enough of the api to make a judgement on the last statement. Although the pathfinding API seems powerful if it just works in a call -> result fashion.”
- “Estimating lines of code for implementations of rough descriptions of requirements has a highly random outcome.”

A.3. Wilcoxon Rank-Sum Test Results

[1] "How many lines of code do you think you would need to implement your own path finding algorithm?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 249, p-value = 0.00305
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement your own decision trees?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 175, p-value = 0.03741
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement your own state machines?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 147, p-value = 0.07865
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the seek steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 189, p-value = 0.06418
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the flee steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 187.5, p-value = 0.07087
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the pursue steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 191.5, p-value = 0.05424
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the evade steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 196.5, p-value = 0.03771
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the arrive steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 200.5, p-value = 0.02766
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the wander steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 187, p-value = 0.07336
alternative hypothesis: true location shift is greater than 0

[1] "How many lines of code do you think you would need to implement the flocking steering technique?"

Wilcoxon rank sum test with continuity correction

data: own_implementation and vgail_implementation
W = 191, p-value = 0.05621
alternative hypothesis: true location shift is greater than 0