Toni Lääveri

Integrating AI for Turn-Based 4X Strategy Game

Helsinki Metropolia University of Applied Sciences

Master's Degree

Information Technology

Master's Thesis

29 September 2017

Helsinki Metropolia University of Applied Sciences

Preface

This postgraduate study was done as part of the Master of Engineering Degree Program in Information Technology for the Metropolia University of Applied Sciences. The motivation for this thesis originated from personal interest in both the 4X strategy games and the game artificial intelligence in general, having experienced this genre first time when playing Sid Meier's Civilization back in 1993 on the Macintosh LC II computer.

While I have been working in the game industry for over a decade now, the possibilities for exploring this particular area of interest have been limited, so the possibility of using it as the topic of my thesis was a rather natural choice. Although the scope of the research was very broad and the results did not reach the practical levels which I hoped for, working on this project was highly rewarding and will further motivate me for years to come.

I would like to thank my supervisor Ville Jääskeläinen, my family and friends for their support and encouragement during the writing of this thesis.

Toni Lääveri
Helsinki, 29.9.2017

Author(s) Title Number of Pages Date	Toni Lääveri Integrating AI for Turn-Based 4X Strategy Game 100 pages + 6 appendices 29th September 2017
Degree	Master in Engineering
Degree Programme	Information Technology
Instructor(s)	Ville Jääskeläinen

Although computer games have been around for over half a century, the gaming has matured to become a mainstream phenomenon in the past decade, partly propelled by the breakthrough of mobile platforms which provide users access to a vast selection of games. As the complexity and selection of games is constantly increasing, there is growing pressure to provide meaningful artificial intelligence (AI) opponents in certain gaming genres.

This master's thesis focused on finding a way to implement an AI player for a turn-based 4X computer strategy game. As there was no suitable project at work to apply this research on at the time of the study, the project was created as a personal venture with a theoretical game used as the source for requirements.

In the research phase, dozens of sources of existing literature and information about the field were analyzed, which included extraction of the knowledge and technologies appropriate for this project. The selected technologies were documented in the thesis, and their use cases were identified, and examples were created for most of them.

A high-level technical design for AI integration was created as an outcome of this thesis, which describes a proposed architecture for the AI opponent and combines the technologies evaluated in the thesis into a modular framework. These features can also be leveraged in the player assistance features such as micro-management automation and advisor functions.

The resulting design was supported by actual prototype development done in Unity Editor of selected key technologies. These prototype implementations included a rule-based system inference engine, A* pathfinding on a hexagonal grid map, a spatial database for tracking map data such as player influence, and tactical pathfinding leveraging the information in this database.

Although a complete game was not created during the project, the technical design and research done can be used as a foundation for building AI opponents in turn-based strategy games in future. The practical implementations will most likely also provide feedback on the possible shortcomings of the design and possibilities for improvement, which will be reflected back on the design.

Keywords	Artificial Intelligence, Computer Games, Game Industry, Prototyping, Strategy Games, Technical Design, Unity

Contents

Appendix 2: Screenshots of Spatial Database Prototype Levels of influence

Appendix 3: Screenshots of Tactical Pathfinding Prototype Modes

Appendix 4: Screenshots of Inference Engine (Editor Mode) Prototype

Appendix 5: Source Code of Pathfinder and Spatial Database Prototype

Appendix 6: Source Code of Inference Engine Prototype

Glossary

4X	EXplore, EXpand, EXploit and EXterminate, a strategy game sub-genre.
A*	A-star, pathfinding algorithm used often in games.
AI	Artificial Intelligence.
BT	Behavior Tree, a decision-making method.
CPU	Central Processing Unit, the computer processor
DT	Decision Tree, a decision-making method.
FSM	Finite State Machine, a computational model used to control for example game logic, behaviors and animations.
GOAP	Goal-Oriented Action Planning, an approach to implementing Goal-Oriented Behavior.
GOB	Goal-Oriented Behavior, method of decision making where the needs of AI are expressed as goals which it wants to fulfill.
GPU	Graphics Processing Unit, a specialized computer hardware dedicated to accelerating 2D and 3D graphics
GPGPU	General-Purpose computation on the GPU, term given for using the Graphics Processing Units to other tasks than rendering
Hex	A cell in hexagonal map, with shape of hexagon, i.e. six-sided regular polygon.
HTN	Hierarchical Task Network, a planning method for decision-making.
ID3	Iterative Dichotomiser 3, an algorithm for generating Decision Trees from sample observation data.
IDA*	Iterative Deepening A*, a variation of Iterative Deepening Search (IDS) algorithm which uses A* heuristic for cost estimation.
IDS	Iterative Deepening Search, algorithm which traces a state-space graph iteratively by increasing search depth on each pass.
NPC	Non-Player Character, an entity in game which is controlled by the computer.

OO	Object-Oriented, a programming paradigm in which code is structured in classes and data is presented as objects which are instances of these classes.
STRIPS	Stanford Research Institute Problem Solver, an approach for implementing Goal-Oriented Behavior from the 1970s.
Unity	Popular game engine and development environment, also known as Unity3D.
QC	Quality Control; person and/or department in software development for testing the quality of created product.

List of Figures

List of Tables

1 Introduction

Since the early days of first publicly available video games in 1970s, the business around gaming has grown to a large and high-revenue business, known as interactive entertainment or video game industry. These are some key facts about the industry mentioned in Entertainment Software Association's annual report of 2015 to give perspective about the business case for this project [1 p. 14]:

- Video game industry generated $23 billion sales in United States alone in 2014
- Gamers spent globally estimated $71 billion on games in 2014
- There are 155 million people in United States that play video games
- A total of 1641 video game companies are in United States alone, spread along 1871 locations
- Education of video game industry is being offered in 496 programs in 406 schools in United States

The process of game production can be simplified as an illustration of the value chain of a production, as seen below in Figure 1:

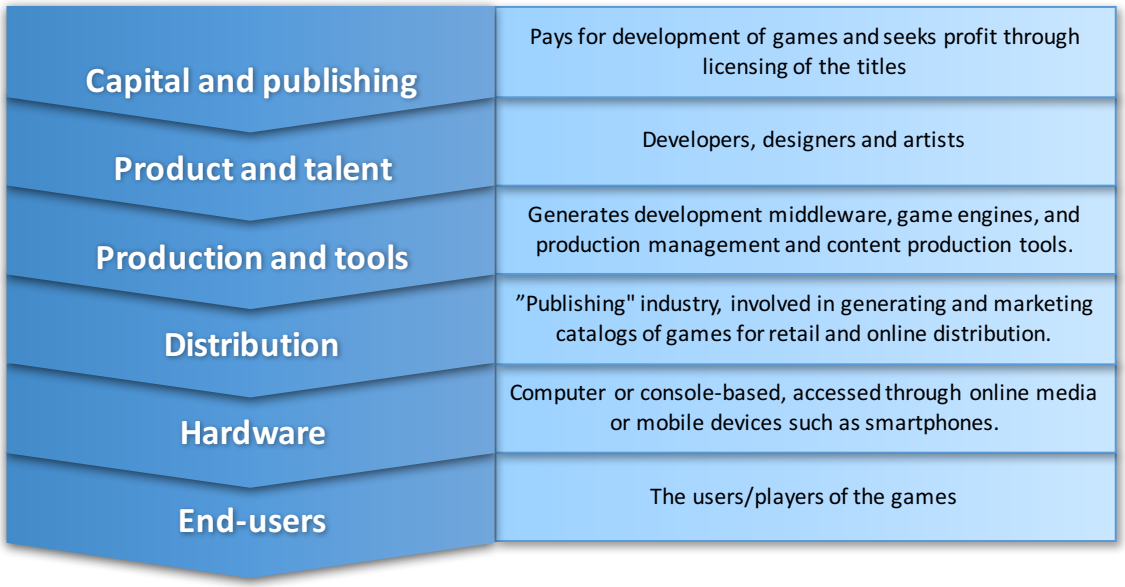


Figure 1. Game industry value chain as defined by Ben Sawyer [2].

The "Product and talent" and "Production and Tools" sections of the value chain are most relevant to the work done in this thesis.

Time Constraints in Production

In game industry, the schedule of game production has a crucial role because it directly affects the development costs of the title. Due to this, there is limited time for doing specific pioneering research during game development, as there is a lot of pressure to meet deadlines and to provide value for the producer. Sometimes, there are separate teams which will do engine and technology development, but even in that case prioritization needs to be done for optimal developer resource allocation. In the author's experience, Artificial Intelligence (AI) development often ends up being one of the fields which will get less focus [3].

Role of AI Programming

Because of the previously mentioned factors, especially in small and medium-sized teams there usually is not a dedicated AI programmer, but those features will be rather added into the game by gameplay and generalist programmers. This allows better flexibility in the team resource management, and usually generalist programmers do have good basic knowledge on the topic to create usable and functional implementations. However, when the requirements for the AI features and quality increase, dedicated skills in AI programming will prove to be invaluable.

Comparison between Academic AI research and Game Industry

When people talk about AI, they are generally referring to the academic AI which is quite different from the game AI. While academic AI aims to solve problems requiring intelligence, the purpose of game AI is to give illusion of intelligence and entertain the player. The goal of AI research in academic setting is usually to create publications and do original research, while game AI development aims to create a game. Academic AI research is often funded by grants, academic institutions or sponsorships, while game AI is funded by the game publishers. It's said that there is strong division between the two fields, but in practice both parties have the possibility to benefit from each other; with help of the results of academic research, game developers can add increasingly advanced AI to the games, while academic AI research can benefit from game engines which they can use for their research [3].

1.1 Scope of Thesis and Research Design

The goal of this thesis was to produce a technical design for adding AI features into a turn-based strategy game. This included:

- Figuring out the requirements imposed by the game for the AI.
- Determining which of the existing solutions in the field of AI knowledge were appropriate to satisfy those goals.
- Designing the best methods for integrating them in the game.
- Choosing proper balance between AI programmers' and game designers' workload by necessary tools.

A full, working game where the technical design would have been implemented was out of the scope of this thesis, but limited amount of prototyping was set as a secondary goal during the writing process to provide some practical analysis on the feasibility of theoretical choices made in the thesis. The research design is illustrated in Figure 2.

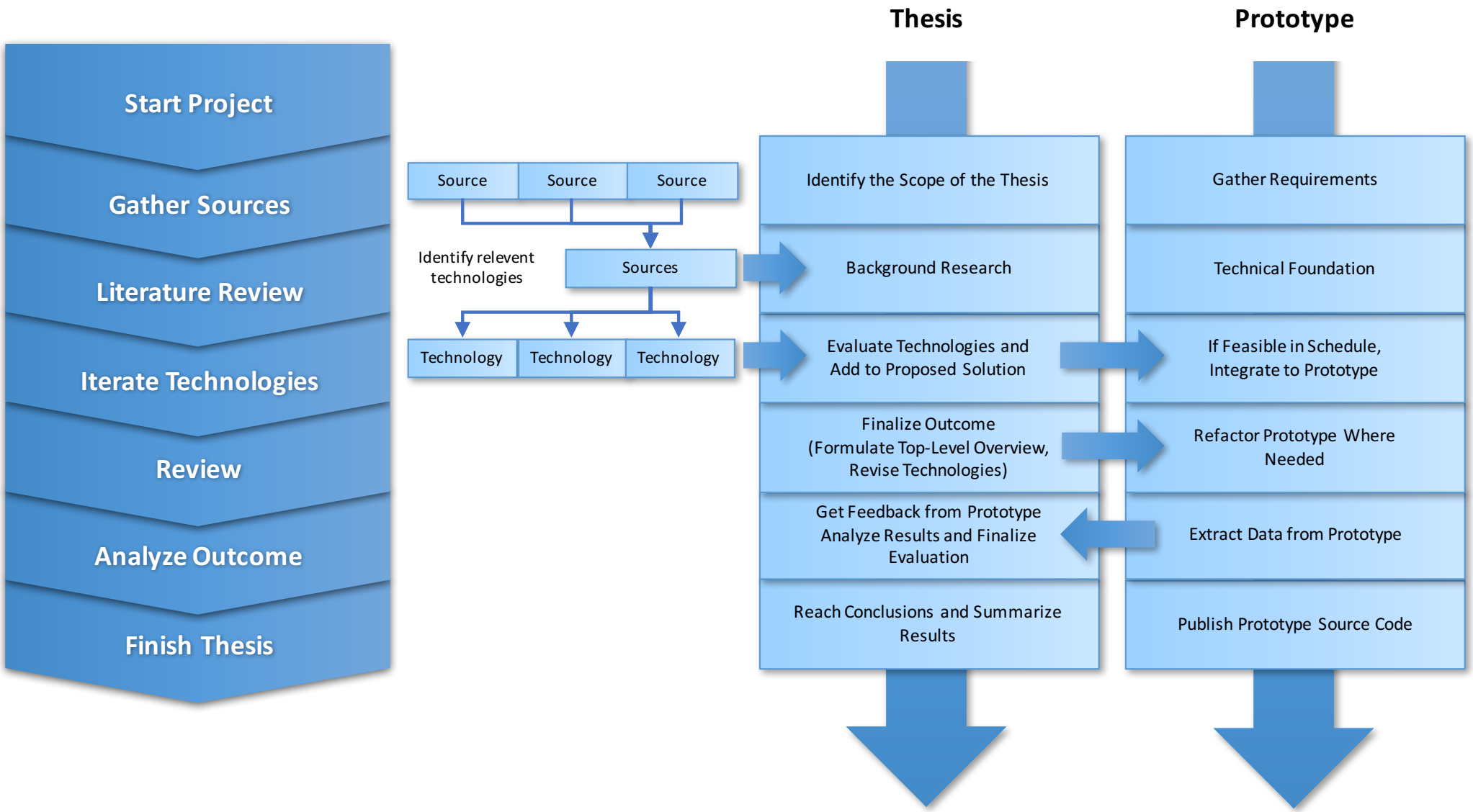


Figure 2. Research design of the thesis.

The research design (Figure 2) shows the phases of the project and the respective thesis and prototype work involved in each of them. Most of them match the chapters in this thesis, in the following order:

1. Introduction chapter outlines the goals and motivations behind the thesis work and gives introduction to the game business to which this work relates to. The initial prototype work was also started at this point, and the requirements it imposes for project were gathered
2. Background chapter contains brief history of AI in games and the related gaming genre, and describes the prototype and lists the previously gathered requirements. During this phase, the relevant technologies were picked
3. Technology chapter was created through iteration of selected technologies, during which the purpose of each one of them was described in the context of this thesis work. Also, some of them were integrated in this phase to the prototype
4. The Proposed Solution was developed initially during the technology iteration phase, and finalized in the review phase. This included outlining the high-level system, during which prototype was also refactored based on the results of earlier integration
5. In Evaluation chapter, the feasibility of proposed solution is evaluated, and this evaluation is partially backed by the data that was extracted from the prototype at this stage
6. The Discussion and Conclusions chapter analyses the results of the thesis, evaluates the outcome of the research and contains suggestions for further improvements

In addition, the project also produced a partially working prototype, for which the source code was included in the appendices of this thesis.

2 Background

This section gives a brief overview of the history of game AI and the 4X strategy gaming genre which the project belongs to. It also describes the technical starting point of the prototype and outlines its requirements for the AI.

2.1 History of Artificial Intelligence in Games

The concept of artificial intelligence itself is nothing new; even in ancient times, mankind has been interested in the concept of artificial life and intelligence. This shows in the various fictional stories and attempts to build automatons even centuries ago. However, the major breakthroughs in electronics and computer technology in past century have allowed unprecedented advancement in research of artificial intelligence [4 pp. 4-5].

Early Applications

After the first general purpose programmable computers were invented, it did not take long time until the first AI programs were developed. One of the first published ones was Alan Turing's chess program, although at the time the computers were not advanced enough to completely run it [4 p. 6].

Although there has been constant academic interest in AI research, breakthrough in game AI started in the 1970's, when first video arcade games were developed. They featured primitive computer opponents such as the aliens in Space Invaders and ghosts in Pacman. Although the AI in those games operated on simple deterministic algorithms, they gave the player impression of intelligent behavior. Since the early video games, the game AI development has slowly advanced, and different gaming genres have their own specialized requirements, ranging from tactical real-time Non-Player Characters (NPCs) in first-person shooters to complex strategic planning in strategy games, and even artificial life simulation [5].

Current State and Future Development

In the past decades, there has been a lot of progress especially in terms of graphics quality and storytelling aspects of games. With the increased complexity and depth of

games turning into interactive entertainment, there is increased desire for creating advanced artificial intelligence functionality to improve the gameplay experience and immersion for the players. Thanks to the constantly advancing computing capabilities of modern computers there are now better possibilities to implement advanced AI than ever before, such as increased focus on self-learning AIs that will adapt and learn from players [6 pp. 3-5].

2.2 4X Strategy Games

The term “4X” was created by Alan Emrich, who used it in review of “Master of Orion” in 1993 for the first time:

> *“...it features the essential four X's of any good strategic conquest game: EXplore, EXpand, EXploit and EXterminate. In other words, players must rise from humble beginnings, finding their way around the map while building up the largest, most efficient empire possible. Naturally, the other players will be trying to do the same, therefore their extermination becomes a paramount concern.”* [7]

Although majority of 4X strategy games are turn-based, there are a few examples of real-time games which are considered to belong to this genre. However, the classification of games in 4X genre can be difficult, and sometimes a few additional criteria have been used to narrow what fits the definition, such as empire economy control and diplomacy versus combat-focused gameplay in regular strategy games [8].

Unlike many other game genres, strategy games are highly dependent on skilled AI to provide meaningful gameplay experience for the player. As the opponents’ tactics and strategy are foundation of the core gameplay challenge for player, a poorly implemented computer AI would most likely be acceptable by casual gamers, but advanced players would find it boring and unrewarding [5].

Civilization

Sid Meier’s Civilization series is one of the best-known examples of turn-based 4X strategy games (see Figure 3). The game has taken heavily inspiration from previous board

and computer games such as Risk and Empire, but also added novel features such as technology tree [9].

Figure 3. The original Sid Meier's Civilization (Macintosh version pictured).

Originally released in 1991, the series has had six major releases and several spin-offs. In Civilization, the player takes role of leader of a nation which he/she will lead from stone age to modern day. In the first game in the series there were only three conditions for ending the game; world domination by eliminating all opponents, building a spaceship to reach Alpha Centauri, or running out of time in the year 2100. Later versions of game have gradually added more victory and endgame conditions along with many other features in each major release [9].

Master of Orion

In Master of Orion (see Figure 4), the players control number of different races which compete for control of the galaxy. The game features exploration, discovery of new technologies, dealing with other players with diplomatic or military means, and endgame conditions which are similar like to the ones in Civilization [10].

Figure 4. Master of Orion II.

Besides being space-themed, the game trades off the grid-based movement and exploration with a more restricted and simpler model where movement is only allowed between solar systems. Other major difference is also how battles are handled; instead of single-unit attacks, the combat happens on a separate combat screen where multiple fleets of both participants are fighting at once. Players have control over individual ships and weapons, which can lead to complex tactical battles [10].

The game has been influenced by some earlier games, such as Reach for the Stars. The original series had three releases, and the franchise was recently rebooted by Wargaming on Steam [7; 10].

Galactic Civilizations

Galactic Civilizations (see Figure 5) is a series of strategy games, which was initially released for OS/2 in 1993 by Stardock Corporation. The game combines the turn-based

grid map familiar from Civilization with space exploration theme like in Master of Orion [11].

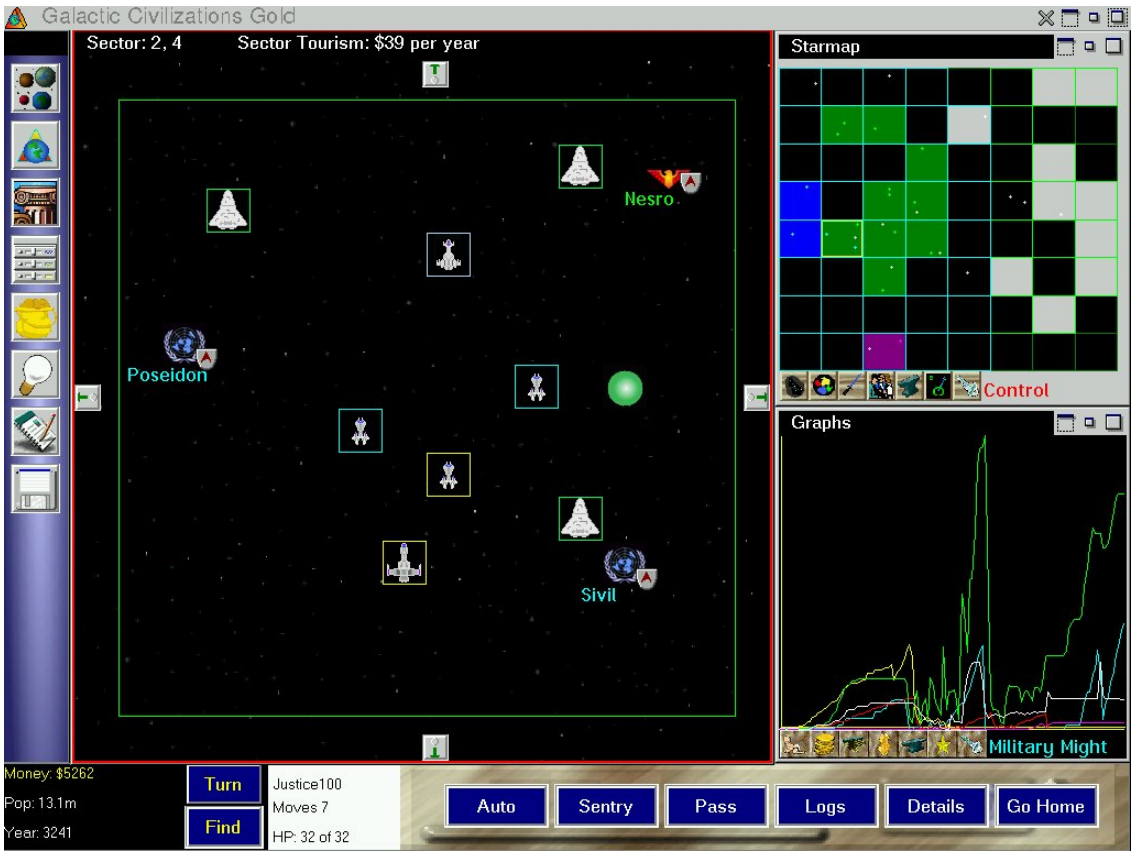


Figure 5. Galactic Civilizations Gold for OS/2 [12].

Galactic Civilizations was one of the few games released for IBM's OS/2 operating system, and it received recognition even from IBM who licensed the game and included it rebranded as "Star Emperor" in their FunPak software bundle. However, the OS/2 operating system had limited user base and later lost its remaining market share to Microsoft Windows on the PC platform, thus later releases were made for the Windows platform, most recently on Steam [11; 13].

2.3 Prototype of Strategy Game as Foundation

Due to time constraints, the implementation part of this thesis focuses on working on a theoretical prototype of the game. This allows the development to focus on areas relevant for artificial intelligence integration.

The game itself is be turn-based, with each player performing actions in sequential order. The game world is represented as a hexagonal grid map, to which players have their own views depending on exploration and espionage status. The game draws inspiration from Civilization and Master of Orion series, creating mix of them with resemblance of the original Galactic Civilizations.

Unity 3D

For rapid prototype development, an off-shelf game engine is used. Unity 3D (see Figure 6) is one of the most popular engines today, which has not only gained popularity as mobile game development platform, but also has seen use in recent AAA-grade desktop titles such as the recent remake of Master of Orion and city-planning simulator Cities Skylines [14].

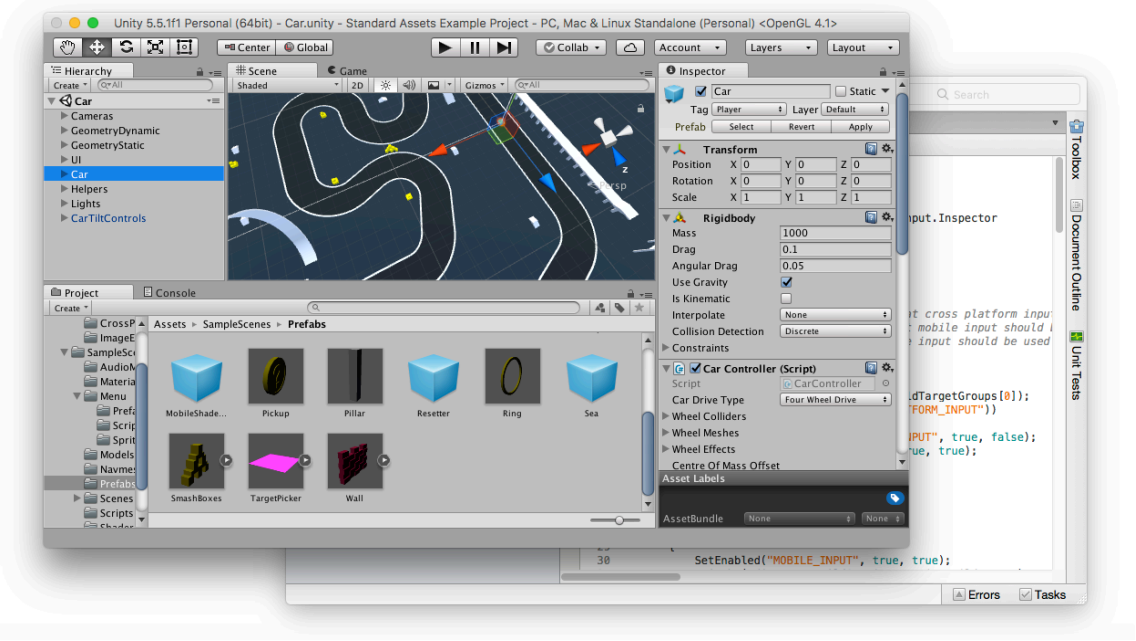


Figure 6. The Unity3D Editor running on macOS platform.

Originally released for Mac OS X platform in 2005, the engine has had five major releases and is currently available for both Windows and Mac OS X [15]. Unity features component-based architecture, advanced 3D engine that can target various graphic APIs on several platforms, NVidia PhysX engine and for physics simulation, and various other frameworks. Gameplay logic in this project is implemented in C#, which is supported as default scripting language by Unity (other option being the UnityScript). It is a

mature and high-level language originally developed by Microsoft, which is integrated into Unity through the Mono framework [16].

The choice of Unity for prototyping does not limit the options for selecting different game engine (or building a custom one) for the final game, but given Unity's track record of being the platform of choice in number of high-quality titles, using it for the actual production is a viable option.

2.4 Requirements

To evaluate the technical needs for AI integration, a set of requirements was created based on the game concept, which outline the expected features of the game. These are high-level non-functional requirements, which also outline the system-level design.

2.4.1 Common Features

There is a set of features which are not specific to AI players, but are also used by human players in the game. A historical challenge for 4X games has been the amount of micromanagement which increases as the game progresses. The amount of micro-management is directly proportional to the size of game world, complexity of game economy, and length of the game. This has potential to frustrate players and when they have to spend excess amount of time dealing with lots of repetitive low-level tasks, instead of focusing in larger scale strategies and planning.

There are various ways to reduce this micromanagement which have been added to recent 4X games, and there are a few ones that considered when defining the requirements. The quality of artificial intelligence to which mundane tasks are delegated to is important, because if a poor implementation makes players feel that automation makes worse decisions than they would themselves do, it would discourage them from using automation altogether and make the attempt to reduce micromanagement void.

Colony management

There are a few aspects in managing colonies which can be delegated to automatic colony governor AI:

- Production management
- Work force distribution
- Tax rates
- Import and export balancing
- Security level

It should be noted that having the support for automatic control of these properties of colony does not prevent the human player from altering or disabling any of them if he/she feels like it. Also, depending on how much configuration will be exposed to the player, any of these could be parametrized with user-specified customization.

Automated units

Another good opportunity to reduce mundane tasks for the human player is the automation of certain unit actions:

- Worker automation (build new improvements and adjust existing ones)
- Automatic exploration (reveal unexplored space and patrol previously explored areas)

The above tasks are good fit to be implemented with AI automation, as the scope of strategic choices made in them are focused on certain isolated parts of the game, and thus do not depend on the high-level strategic plans player may have. The worker automation has the possibility to determine best possible improvement actions based on the economic status of the player's empire and colonies, and can adapt also previously made improvements to match the most recent situation. Exploration is also by itself an isolated function, where different input patterns can be fed to the exploration AI to prioritize areas to explore for example based on evaluation of military threat on influence map.

2.4.2 Computer Player Specific Features

The rest of AI features are specific for the computer player, and they are used to simulate the actions a human player would be performing in the game.

Long-term Planning

The computer player needs to be able to choose and strive for various long-term goals, most important one being desired victory condition. The AI should be able to reach this goal by dividing the plan into smaller sub-plans, and adjusting them to match the constantly changing conditions which are affected also by other players during the game.

AI State Persistency

All the data used by the AI should be serializable, so that the game state can be saved and loaded at any time without disrupting the functionality of the computer player's decision making.

Diplomacy

The AI should have ability to make diplomatic decisions, including declaring war and creating alliances. It should have possibility to do those choices in informed manner, with knowledge about the other player's economic and military power, past trustworthiness, and any other game parameters (i.e. common ideologies, personalities, etc.) that might make the other player more or less favorable.

2.4.3 Optional Features

There is a set of features which should also be evaluated, though they are not required by the core gameplay, and thus can be considered optional. If implemented, they do though have possibility to enhance playability value for more hard-core players.

Tactical combat

When battles are initiated between fleets, they are by default be resolved using simplified simulation which considers only the ship statistics, numbers, combat bonuses and other predetermined factors affecting the battle. This can be enhanced by introducing tactical combat, in which entire battle is fought as turn-based mini-game, and moves of computer players are handled by tactical AI. Human player can either choose himself/herself all moves against the opponent, or activate an automatic battle mode in which the same AI features will fight the battle also for him/her.

Ground combat

The fight over control of colonies on planets is handled through ground combat. By default, the results of these battles will be determined by the ground troop technology level, number of troops, and other possible combat bonuses. There is possibility to further expand this battle into more fine-grained ground combat simulation similar to the tactical combat in space, where players would have more opportunities to control individual platoons of troops. Practically this would be similar to the previously detailed tactical mode and could leverage some of the technologies, but would happen on planet surface instead of space.

3 Technology

Since the early applications of AI in computer games, the number and complexity of technologies involved have gradually grown, partially with help from advancements in processing performance, but also due to research done in the field of artificial intelligence research done for academic purposes. The technologies introduced in this section are chosen by their potential usability in the game being created, allowing the scope of thesis to focus on the most relevant ones.

General Architecture

When creating AI for strategy game, attention should be given to the design of overall architecture; technologies involved, how they are bound together, and how they impact the gameplay. According Ian Millington, turn-based strategy games share many aspects with real-time strategy games, with most important ones being Pathfinding, Decision Making, Tactical and Strategic AI, and Group Movement [17 pp. 809-815] as shown below in Figure 7.

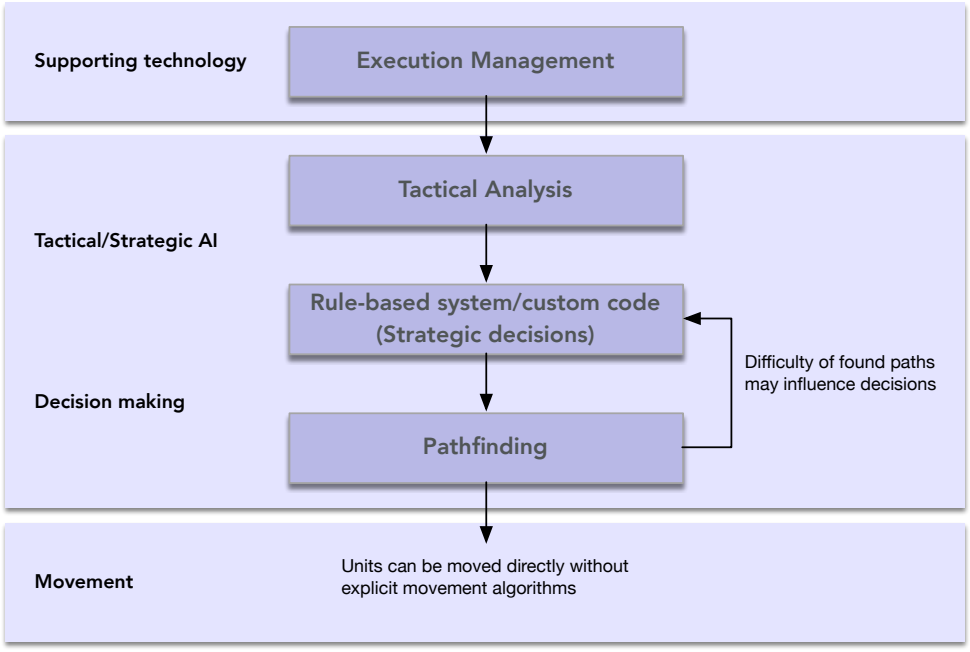


Figure 7. An example of AI architecture for turn-based strategy games [17 p. 815].

Figure 7 shows a possible architecture for turn-based strategy games, although exact model has variations depending on the gameplay elements involved in the design. The

technologies presented in this thesis use that model as a starting point, with adjustments done to suit the exact requirements of this project.

3.1 Introduction to Traditional Board Game Techniques

As previously briefly mentioned in Chapter 2.1, one of the earliest applications for computer game AI was the game of chess. Since those early days, a large amount of time and effort has been spent on research and studying AI for several board games which have provided both academic and practical challenge for the researchers. During the past decades, a set of algorithms has gained foothold to become the foundation shared by many of the board game AIs, and a few key concepts are introduced in this chapter [17 p. 647].

Game Tree

The most important concept for the majority of board games is the game tree, which represents the game states as nodes in the tree, and all possible moves as branches leading from nodes to new possible states. With this data structure, the goal for the AI is to choose one of the branches as a move it should make, and needs to use various algorithms to find out which move is the best one it can make [18 pp. 16-29].

Minimax

When evaluating the game tree, the basic idea is to use a heuristic to give each possible move a score, which indicates how good the move would be for the player; in single-player games, the heuristic could for example be the number of moves to finish a game. The score of each move bubbles back up in the search tree, and after scoring all possible moves the AI just needs to choose the move that has the best score. However, when two or more players are involved, the evaluation algorithm should not only try to find the best score for the player, but also acknowledge that the opponent will try to choose a move that yields the least score for the player. Thus, when bubbling up the scores the minimum score should be picked for enemy moves, and player should choose move which give the largest of the minimum scores, hence the name “Minimax” of the algorithm [18 pp. 30-39]. An example of the basic principle in minimax algorithm can be seen below in Figure 8.

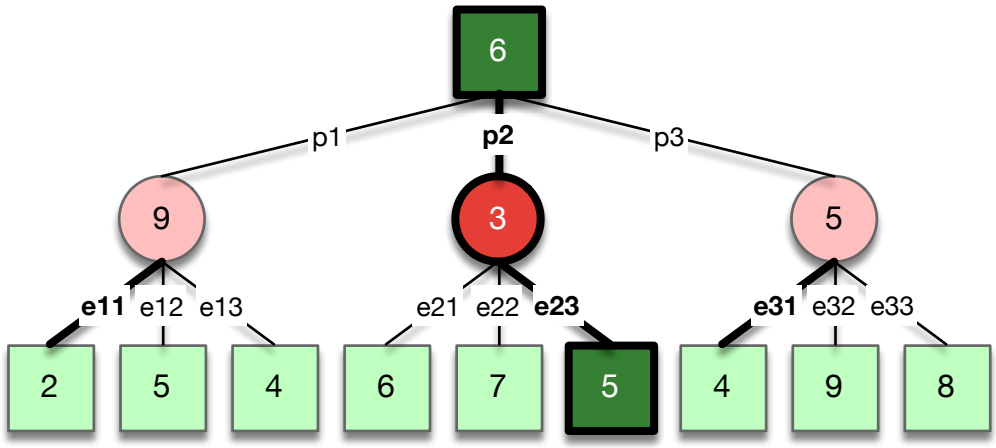


Figure 8. Minimaxing example in a game tree.

In this case, the best move with 2-ply search would be p2, because it would end up with score of 5 for the player. Other moves end up with lower score, because move p1 would allow opponent to do move e11 resulting with score of 2, and p3 would allow move e31 with score of 4, both of which are lower than the smallest score of 5 given by move e23.

The challenge with game trees is the balance between ply depth and processing power requirements; the deeper the tree is, the number of possible moves usually increases exponentially, and so does the time required to process it. On other hand, if the search depth is too short, the AI cannot predict the game events far enough in advance, and might not be able to predict possible “killer” moves which might decide the winner of the game in long run. There are various methods which have been developed to help and speed up searching the game trees, including Alpha-Beta Pruning, Killer Heuristic, Alpha-Beta Windows, and Transposition Tables [18 pp. 40-60; 17 p. 651]

Applications in turn-based strategy games

Although there are similarities between board games and strategy games, the complexity and number of possible moves during each turn in strategy games causes the size of game tree to grow so large, that using traditional board game AI algorithms such as minimax becomes unfeasible in most situations. There are however certain cases in which using some of the techniques are beneficial, such as using cost estimation similar to the game tree scoring approach with task planning which can be used for example in decision making for research, construction, troop movement and military actions [17 pp. 688-670].

3.2 Virtual Player

Traditionally the AI has been implemented in games through the concept of AI agents, which are usually divided in two types: Characters and Virtual Players. Characters appear as visible entities in the game that player can usually interact with; they be as simple as ghosts in Pacman, more advanced enemies like demons in Doom, or other NPC opponents. They can either be directly involved in the gameplay, or just exist to add to the ambience and general immersion of the game [4 pp. 11-12].

Virtual Players on other hand do not usually have physical representation in the game world, but instead replace other human opponents in game and assume the tasks and responsibilities of that player. The classic example for this kind of AI is used in board games such as Chess, where the Virtual Player decides which moves it should make in the same way as a human player would. This model applies to turn-based strategy games, where the individual units in the game do not usually contain any intelligence, but all decisions are made by the Virtual Player. Exception to this are real-time strategy games, where the behavior of individual units has important role in the game, but that subtype of strategy games is out of the scope of this thesis [4 pp. 16-17].

For this thesis, the design of the Virtual Player is the central outcome of the research project, as it is an umbrella concept that encapsulate all the supporting technologies studied in this project.

3.2.1 Multi-Tier AI Framework

Building on the principle of the strategy game AI architecture shown earlier in Figure 7, the multi-tiered AI framework approach is based on separating responsibilities of the AI to individual levels. The structure framework resembles the military hierarchy, in which high-level AI sets the general strategy and goals, which translate to commands that are given to the next level, which set their own goals to be able to fulfill those orders. This continues until the individual unit level is reached, where the commands are turned into actions performed by the units [17 p. 544]. An example of this top-down command hierarchy is shown below in Figure 9.

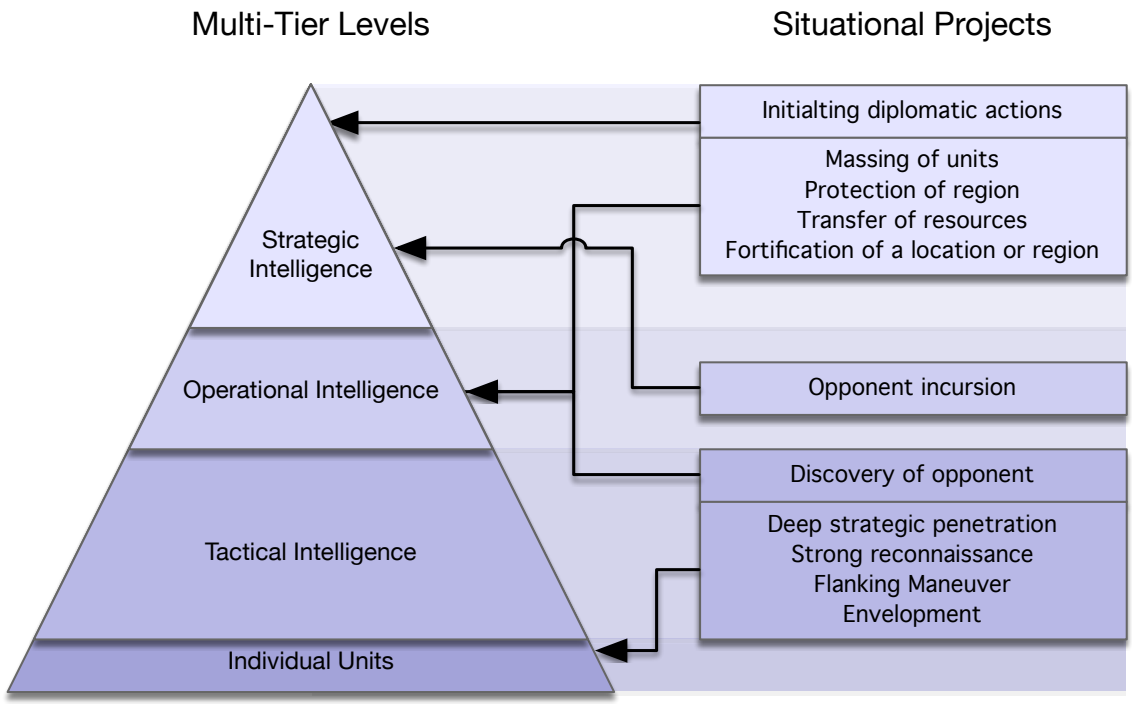


Figure 9. Example of multi-tier AI for military strategy game [19].

In the example, some possible situational projects are shown on their source level with destination visualized. The roles of these individual levels used in the model are described below in Table 1.

Table 1. Roles of levels in the multi-tiered AI model [19].

Level	**Role**
Strategic Intelligence	Knowledge of the entire empire, management of grand strategy, goals, global resource levels, research and diplomacy
Operational Intelligence	Divided into activity groups, can track also non-combat activities such as economic and diplomatic operations
Tactical Intelligence	Information about encountered opponents, geography, resources
Individual Units	Pathfinding, unit movement, combat

There are also other possible ways of building the hierarchy in the framework, including bottom-up approach, where individual units are autonomous and higher levels of the hierarchy just provide intelligence and general information about the game world. It should be noted that the framework allows information anyway to pass in both directions

in the model, depending on how the gameplay requirements imposed on the AI [17 p. 544; 19].

3.3 Pathfinding

One of the most important features in nearly any game dealing with a map with entities that should be able to navigate on it is pathfinding. There are various maps and grids that can be considered to be node graphs as seen in Figure 10, to which graph search algorithms can be applied to. Of the examples shown, hexagonal grid is used for presenting the game world in this project.

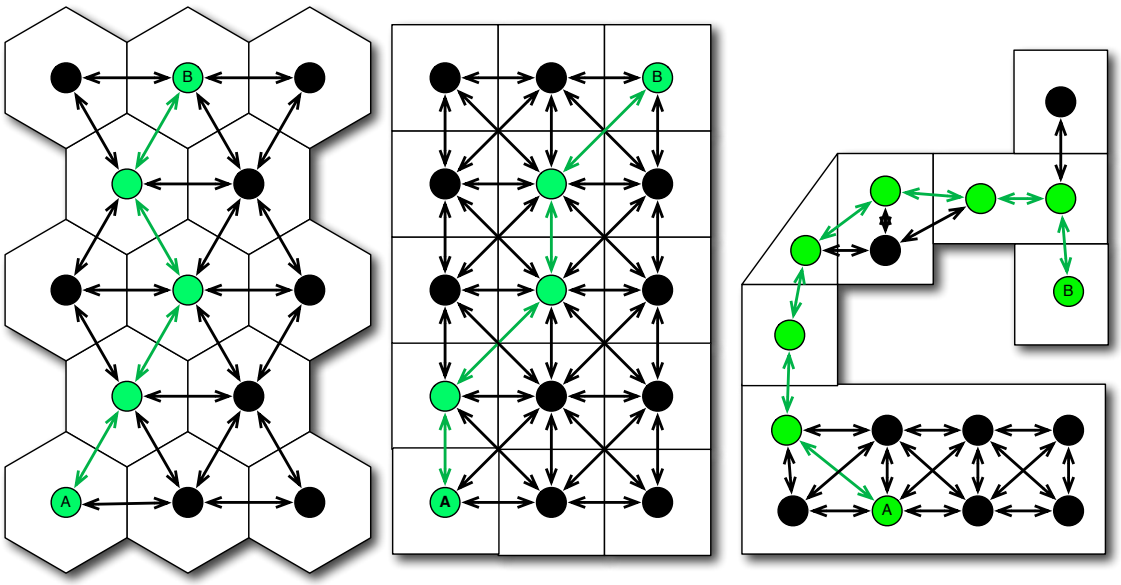


Figure 10. Examples of node graphs created from maps for pathfinding.

The basic case is the ability to find shortest route from point A to point B. To do this, there are several graph and tree search algorithms, the most common ones used in games are summarized in Table 2.

Table 2. Comparison of pathfinding algorithms [4 p. 171].

Algorithm	**Description**	**Benefits**	**Drawbacks**
Breadth first search	Simple algorithm, but does not consider movement cost.	Most simple path-finding algorithm	Not optimized Does not always return shortest path
Dijkstra's algorithm	Improvement on the breadth first search which adds path cost	Guaranteed to find shortest path	Not optimized
A-star (A*)	Combines Dijkstra's algorithm with a case-specific heuristic value to the cost estimation function.	Guaranteed to find shortest path Heuristic helps optimize the search	No major drawbacks

There are numerous other algorithms, but due to simplicity this study only considers the commonly used ones shown in the above table. Because the maps in the game are mostly generated at runtime, have a large number of cells and are highly dynamic, the possibility to precompute navigation data is limited.

3.3.1 A* Algorithm

The A-star (A*) algorithm is one of the most used pathfinding algorithms in games, as it is relatively simple to implement, and has good performance. Because of this, it was chosen as the default pathfinding algorithm for the game. An example of A* pathfinding case is shown below in Figure 11.

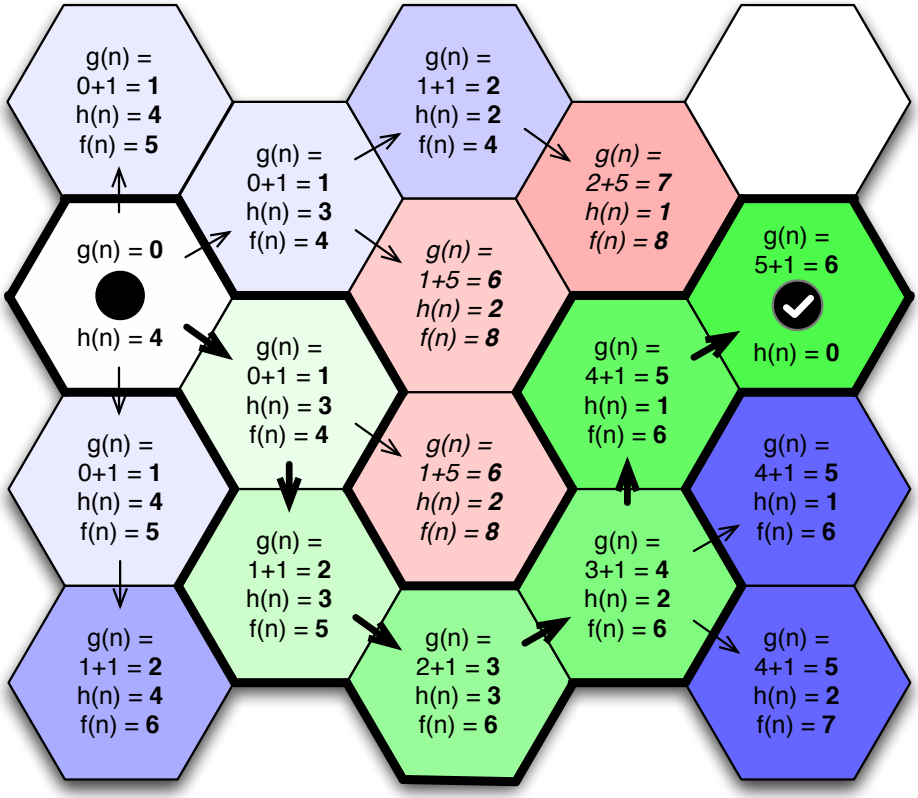


Figure 11. A* Pathfinding on hexagonal grid with Manhattan-distance heuristic.

The pathfinding starts from the hex on left-hand side of figure with black dot, and target cell is indicated with checkmark on right-hand side of the grid. Red color indicates cells with higher movement cost of 5, while other cells only cost 1 to move through. The search begins by putting hex coordinate of starting cell into the priority queue. During each iteration, the first item is removed from the priority queue (one with highest priority), priority is calculated for each neighbor cell which has not yet been processed, and each of them are added into the queue. To calculate the priority, the A* algorithm uses priority formula shown in Equation (1) [20] below:

$$f(n) = g(n) + h(n) \tag{1}$$

In which f(n) is the resulting priority, g(n) is movement cost for this specific cell, and h(n) is the additional A* heuristic value. In this example, the Manhattan distance to the target cell is used. After all neighbors have been inserted to the queue, the iteration proceeds to next step. The search ends, when either the target position has been found as one of the neighbors, or when the priority queue runs out of items (i.e. when there is no solution). In the above figure, green cells (outlined) show the resulting optimal path to target

position. It should be noted, that if the heuristic function h(n) = 0, then the search behaves equally to Dijkstra's algorithm, which A* was extended from [20].

3.3.2 Optimizations

In some cases, the number of search nodes from which pathfinding lookup is queried may end up being too sparse, causing an excessive amount of time being spent on performing the query. In this case optimization is needed, and there are a few approaches that may be beneficial, depending on the use case.

Hierarchical Pathfinding

To speed up searching paths in large sets of nodes, it may be possible to combine physically adjacent nodes as groups, so that pathfinding operates initially on the higher-level group nodes, from which it progresses to lower levels after high-level path is found. Depending on the type and size of original node tree (for example, in large open-space maps), this combining of node clusters can be extended further to higher levels, creating a hierarchy of search trees, which is base idea in hierarchical pathfinding [17 p. 265].

Zone Mapping

In some cases, there may be search trees which have completely disjoint start and goal nodes. When this happens, a A* search would end up having to look through all connected nodes in the tree just to find out that there is no solution for the path query. In zone mapping, a special flood-fill algorithm is used to identify isolated regions in the search tree, with result of this process stored in a zone map. With this cached data, it is possible to know in advance whether there is any solution before having to attempt doing the path query [4 p. 197].

3.4 Decision Making

One of the key requirements for the AI is the ability to make decisions, which translate to actions executed by the computer player. The relationship between input data and action request effects is visualized below in Figure 12.

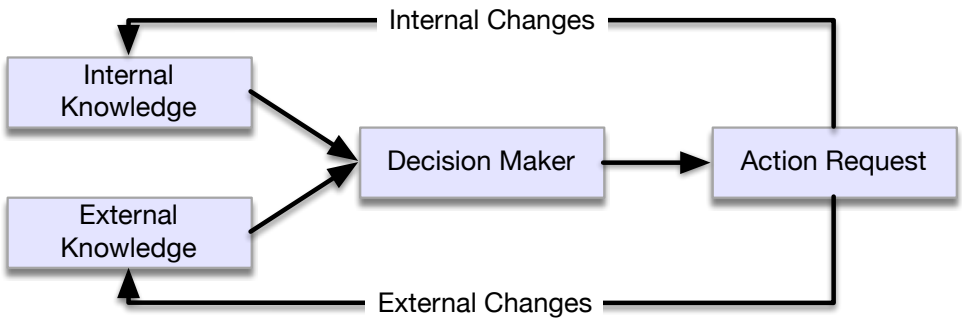


Figure 12. Decision making schematic by Ian Millington [17 p. 303].

There are several algorithms and techniques for this purpose, which share the core idea of having internal or external knowledge sources as input, which are processed to action requests as output which affect the internal or external state of the game [17 pp. 301-302]. This section introduces the most common decision-making methods, which have applications on multiple levels on the Multi-Tier AI model.

3.4.1 Finite State Machines

Finite State Machines (FSMs) are one of the key concepts used in computer games. A state machine is composed of a set of states, and rules defining transitions between those states. In a FSM only one state is active at a time, and switching to other states only happens when one of the transitions out of current state is requested [6 pp. 165-166]. A simplified FSM for scout unit is shown below Figure 13.

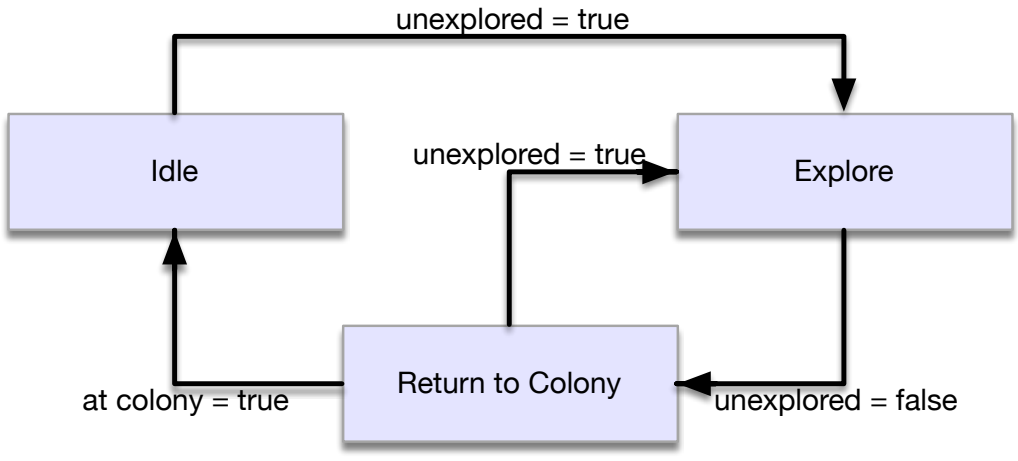


Figure 13. A possible FSM for controlling scout behavior.

There are three states in the FSM to either idle, explore or return to nearest colony, and four transitions controlled by two external flags which indicate whether there are areas to explore, and whether the unit is at colony.

In the ideal FSM model, transitions are handled internally using the data provided to the FSM, and are thus self-contained. However, in some cases using the input data as sole trigger to state changes might not be enough, for example when UI triggers user input which needs to immediately alter the state of a unit. In this situation, an external transition can be triggered procedurally, which is used to set the state of FSM directly without use of a predefined transition condition [4 pp. 50-52].

Hierarchical State Machines

The Hierarchical State Machines (HFSMs) extend the basic principle of FSMs by adding the possibility of using sub-FSMs, which are basically state machines nested inside of a parent FSM. They add flexibility to the state transition options through the ability to continue parent FSM flow after finishing execution of the sub-FSM. Another benefit is the possibility to split more complex states into sub-states [17 pp. 327-330]. Figure 14 below shows a simple example case for Hierarchical FSM.

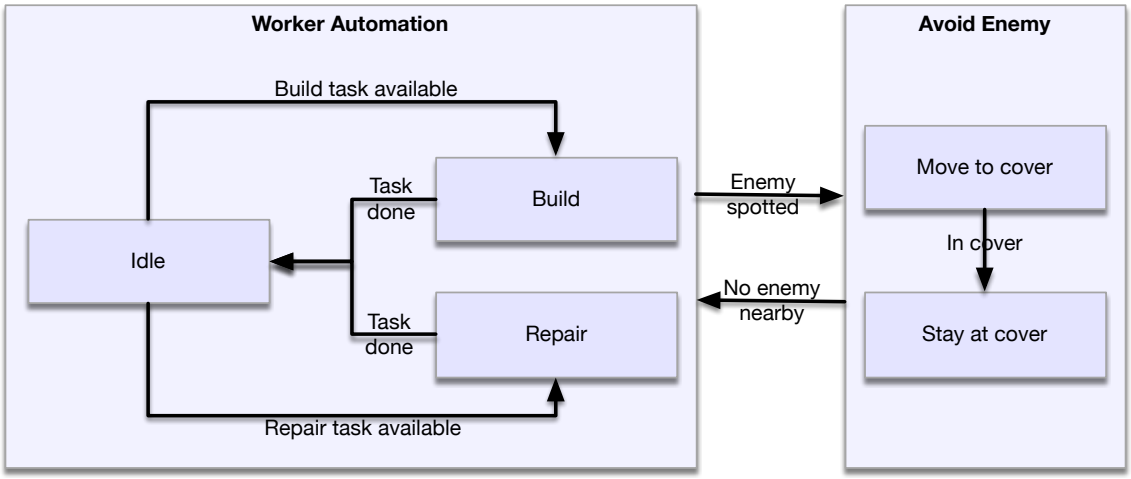


Figure 14. Example of hierarchical FSM for worker unit.

In this case, a worker unit would by default toggle between idle state and build and repair tasks, but at the appearance of enemy unit would interrupt any task it was doing, and seek cover. After the threat of enemy would be cleared, the Hierarchical FSM would resume the worker automation Sub-FSM, and its active state would automatically be the one which was interrupted earlier.

3.4.2 Decision and Behavior Trees

There are a couple of common techniques which have the advantage of both being simple to implement and to use; Decision Trees and Behavior Trees. They are usually used to control NPC actions in games, but they have also potential use cases for decision making in military units in strategy games. They are both tree-like structures, which have the benefit of being able to be shown as a visual presentation of the decision-making process to the AI designer [17 pp. 303-309; 21].

Decision Trees

The Decision Trees (DTs) help making choices based on the world state at a specific time through the use of tree built of decision branches, which lead to actions. Usually the decision nodes are binary and have only two branches, but it is possible to make selections with more than two options. The decision-making process starts from the root, and simply just moves down to the next branch depending on the outcome of each decision node. This flow is very similar to traditional if-then-else control flow in high-level programming languages, but the nodes as usually expressed explicitly as data structures, which can be defined either in code or using data model which the AI designer can modify [17 pp. 303-309]. The above Figure 15 below shows how the previously proposed FSM logic in Figure 14 might be converted into a DT.

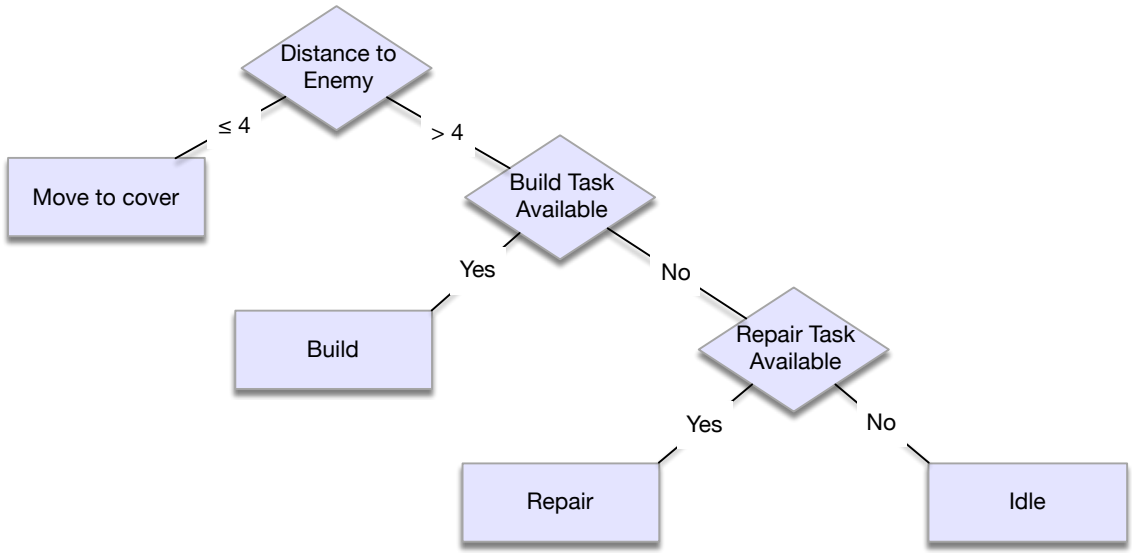


Figure 15. The decision logic from Figure 14 converted into a decision tree.

At root decision, the presence of enemy nearby would be checked first, in which case the worker unit would move to cover. If no enemy were nearby, the tree evaluation would continue to next decisions to check if either type of tasks would be available for it; if not, the idle action would finally be picked as the last option.

Decision Tree Learning

One interesting aspect of DTs is that they can be generated using machine learning from input of observation and action sets, which represent the desired outcomes for each world state observed. There are various possible methods for accomplishing this, but the commonly used ones are based on the Iterative Dichotomiser 3 (ID3) algorithm. It uses the measurement of entropy to calculate information gain from available attributes for selecting the most relevant decision factor as the next node in the decision tree, and continues this process until all action nodes have been created [17 pp. 593-597]. For example, the following observations could be used as input for the ID3 algorithm:

Table 3. Input observations for ID3 algorithm.

Build Task Available	Repair Task Available	Enemy Near	Action
Yes	Yes	Yes	Cover
Yes	Yes	No	Build
Yes	No	Yes	Cover
Yes	No	No	Build
No	Yes	Yes	Cover
No	Yes	No	Repair
No	No	Yes	Cover
No	No	No	Idle

In each iteration, the algorithm first uses Equation (2) to calculate the entropy of the entire set of actions [17 pp. 593-597]:

$$E = -\sum_{i=1..n} p_i log_2 p_i \quad (2)$$

In first iteration, the formula gives the following entropy values for the entire set and available subsets:

$E_s = 1.75$

$E_{havebuildtask} = 1$ $E_{nobuildtask} = 1.5$

$E_{haverepairtask} = 1.5$ $E_{norepairtask} = 1.5$

$E_{enemyneary} = 0$ $E_{enemynear} = 1.5$

Those results can now be used in Equation (3) to calculate the information gain from the subsets [17 pp. 593-597]:

$$G = E_S - \sum_{i=1..n} |S_i|/|S| \times E_{S_i} \tag{3}$$

Which results in the following information gain values for the attributes:

$G_{havebuildtask} = 0.5,$

$G_{haverepairtask} = 0.25,$

$G_{enemynear} = 1$

With the above results, the algorithm chooses the attribute with highest information gain value as input for decision node, which in this case would be the presence of a nearby enemy. After this the algorithm repeats the same process for the both subsets of the observation data, adding new child nodes until all relevant branches have been created. The DT which was created by ID3 algorithm from the sample observations is shown below in Figure 16.

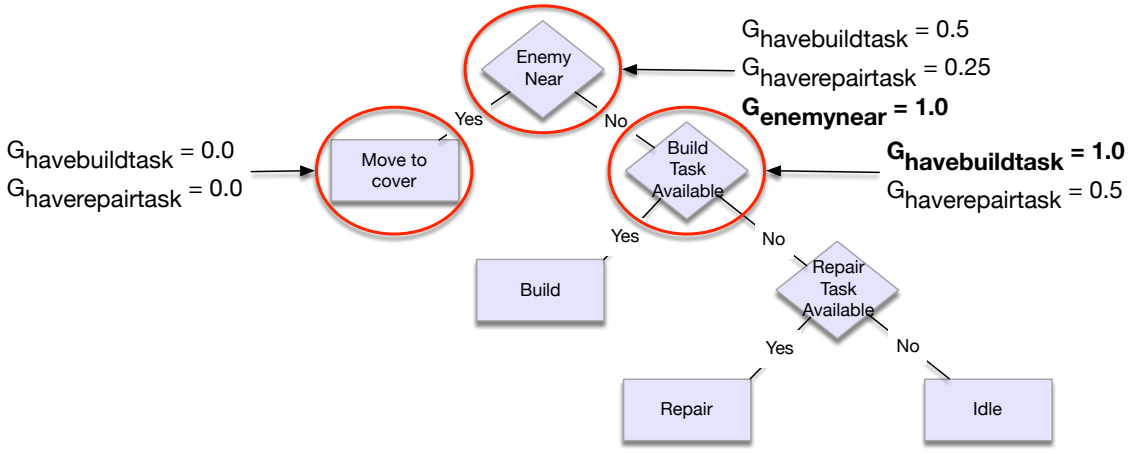


Figure 16. Decision tree generated by ID3 algorithm.

The nodes outlined with red circles were the ones where calculation of information gain was performed. Note that no decision node was added for the “Yes” branch of “Enemy

near" check, because neither of the two remaining attributes contributed any information gain to the decision, and thus were not required at all.

Some software frameworks even include Decision Tree Learning as part of their feature set, for example Apple offers built-in support generating decision trees using machine learning in their GameplayKit framework on iOS, macOS and tvos platforms as part of their basic Decision Tree implementation [22].

Behavior Trees

Another graph-style decision-making method is Behavior Tree (BT), which generalizes the previously introduced Decision Tree concept. Any DT can be represented as BT, but the modularity and extensibility combined with their simplicity is what makes them powerful [17 pp. 52-53; 23]. The difference between DT and BT trees is shown below in Figure 17.

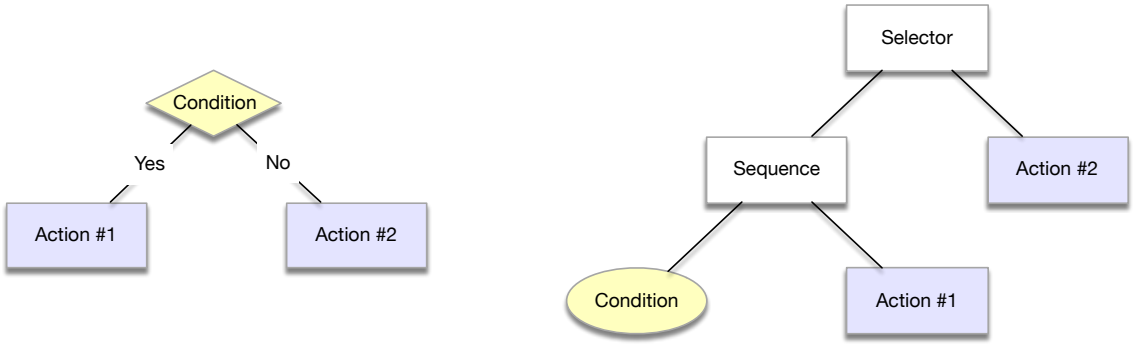


Figure 17. A Decision tree expressed as behavior tree with identical logic [23].

Unlike DTs which are solely composed out of decision and action nodes, BTs have a multitude of possible node types, which are generally divided into interior nodes (also known as composite nodes) which have one or more children, and leaf nodes which have no child nodes. The child nodes in BT are ordered in priority order (usually visualized from left to right), which dictates the evaluation order of the nodes. The processing starts from root node, and progresses down to child nodes in the tree depending on the node types in the tree. All nodes have a precondition which can have three possible return values; success, failure and running. The most commonly used nodes in BTs and their return values are listed below in Table 4 [23].

Table 4. Most common behavior tree node types [23].

Node	**Type**	**Precondition return values**		
		Success	**Failure**	**Running**
Action	Leaf Node	Upon completion	When cannot complete	During completion
Condition	Leaf Node	If true	If false	Never
Sequence	Interior Node	If all children succeed	If one child fails	If one child returns Running
Selector	Interior Node	If all children succeed	If all children fail	If one child returns Running
Parallel	Interior Node	If ≥ M children succeed	If > N – M children fail	If neither Success or Failure condition is met

Each time the BT is ticked, the tree is traversed down and the node preconditions checked until the active task node is reached, which performs the action in during this particular tick. The return value is back-propagated up in the hierarchy to the parent nodes, which depending on their behavior decides what to do (i.e. possibly evaluate next child in interior nodes), and return the appropriate value back to their parent node. This progress continues until the return value reaches the root of tree, which is returned to the original caller [17 pp. 52-53; 23].

Thanks to the flexible structure any type of nodes can be added to the BT - for example, Utility-Based Systems can be leveraged to create a utility selector node, which can use internally utility scoring to choose the appropriate child to execute [24]. Also, reusable parts of BTs can be shared as sub-trees, reducing amount of work needed to create duplicate behaviors.

3.4.3 Fuzzy Logic

The traditional computer logic is based on Boolean algebra, which infers absolute values of either true or false as the only possible conditional states. There are however certain cases in which a more fine-grained evaluation is required, for example to assess the threat of an enemy fleet and choose appropriate actions based on the analysis of the situation. For this one fuzzy logic can be used, in which the absolute true and false states are replaced by a membership degree, which is expressed as normalized value between

0.0 and 1.0 [17 pp. 344-345]. The overview of how fuzzy logic is used is shown below in Figure 18.

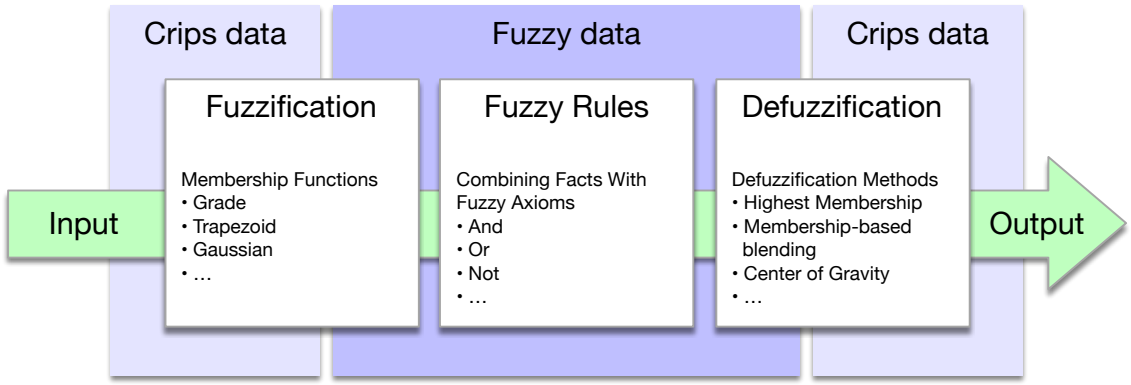


Figure 18. Fuzzy process overview [6 p. 192; 17 pp. 344-354].

The process starts with fuzzification of input data, after which fuzzy rules can be applied to the fuzzy sets, and results can be obtained through defuzzification which converts the data back into crisp values [6 p. 192].

Fuzzification

The Fuzzification is done through of membership functions which convert predefined ranges of values into fuzzy set membership degrees. Commonly used membership functions include grade, reverse grade, triangular and trapezoid functions, but other functions can also be used if necessary. The number of membership degrees is not bound by the number of inputs, as same input values can be assigned to multiple membership sets at the same time [6 pp. 193-198].

Fuzzy Rules

After conversion to Fuzzy Sets, the membership degrees can be combined using Fuzzy Rules which are built using Fuzzy Axioms, which resemble the operators used in the traditional Boolean logic [6 pp. 200-201]. The most commonly used operators are listed below in Table 5. Comparison between Boolean and Fuzzy Logic operators:

Table 5. Comparison between Boolean and Fuzzy Logic operators [6 p. 200].

Operator	**Boolean**	**Fuzzy Equation**
AND	$A \wedge B$	$MIN(m_A, m_B)$
OR	$A \vee B$	$MAX(m_A, m_B)$
NOT	$\neg A$	$1.0 - m_A$

Defuzzification

After combining the values using Fuzzy Rules, the membership degrees need to be extracted from the Fuzzy System to be usable in the game. There are few commonly used methods for doing this:

- Highest Membership
- Membership-based Blending
- Center of Gravity

The best method to use depends on how the data is used; For a simple Boolean decision, the Highest Membership method should be enough, but if there is need to aggregate output strength, other methods are needed. Although the Center of Gravity is often favored, it comes with increased overhead due to the need to integrate surface areas of the membership regions. The blending approach usually is good enough and is much quicker to use [17 pp. 347-351].

Use case: Threat Assessment

One good application of Fuzzy Logic in strategy games is using it for threat assessment and classification purposes. A simple example case for this is shown below in Figure 19:

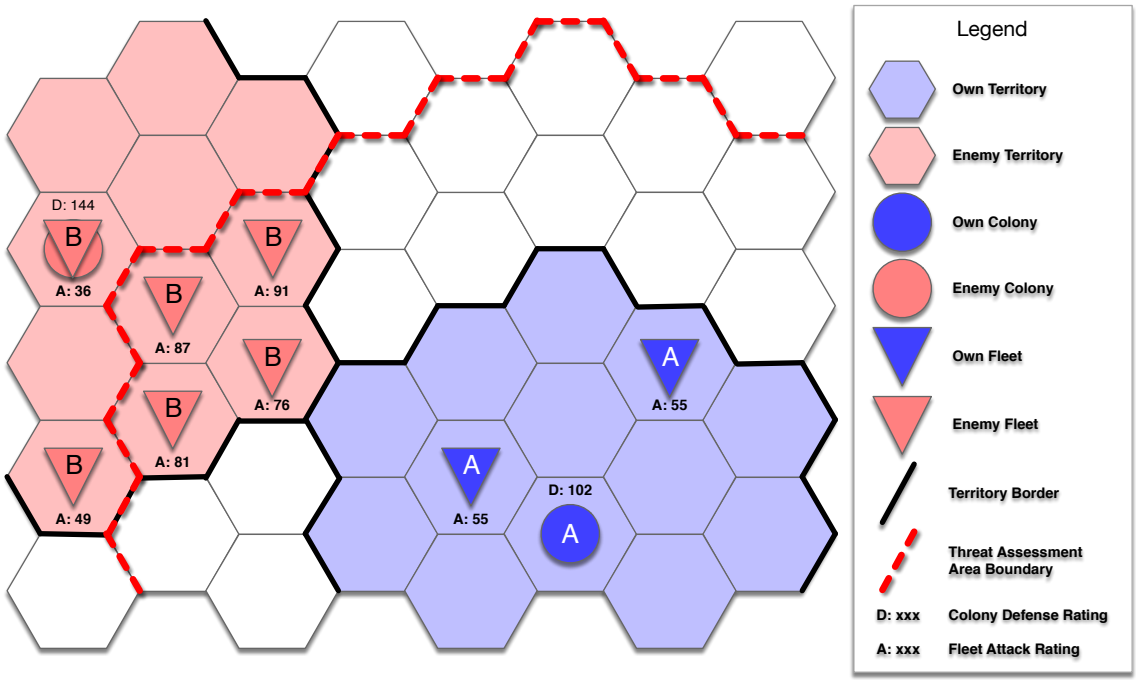


Figure 19. Threat assessment example case for Fuzzy Logic.

In the above case, the AI wants to evaluate the presence of enemy fleets around its colony to adjust its defensive stance if needed. As input data, it uses the proportional ratio between its defensive strength and strength of enemy fleets in Equation (4).

$$p_{ratio} = \log_2 \frac{\sum_{i=1}^{|A_e|} a_{e_i}}{\sum_{i=1}^{|A_o|} a_{o_i} + d_o} \tag{4}$$

Where A_e is the list of enemy fleet attack strengths, A_o is list of own fleet attack strengths, and d_o is the defensive strength of the colony. With the use of $\log_2 n$ in formula, the exponential change in ratio between own and enemy strength can be converted into a linear value which the threat assessment can more practically be applied to. When considering all units within 2 hexes distance from the own territory borders, applying the situation in Figure 19 to the previously introduced Equation (4) results in the following values in Equation (5).

$$p_{ratio} = \log_2 \frac{91 + 87 + 81 + 76}{55 + 55 + 102} \approx 0.66 \tag{5}$$

To use this input value, two membership functions with three fuzzy sets in each are defined as shown below in Figure 20.

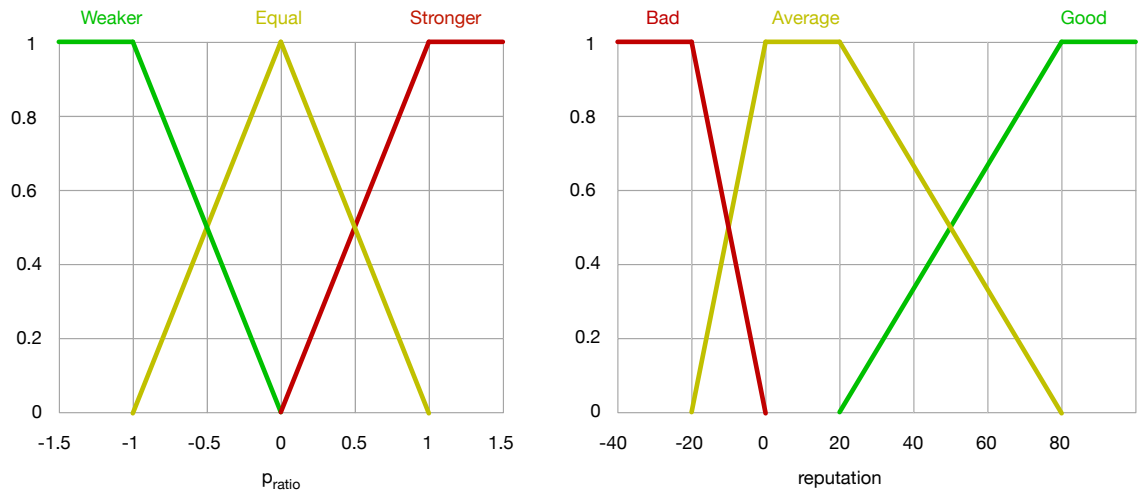


Figure 20. Membership functions for force strength ratio and diplomatic reputation.

The three fuzzy sets in the p_{ratio} membership function evaluate the threat, which is considered to be minimal when the enemy fleet strength is less than 50% (p_{ratio} = -1) of defensive strength, and very high when it is over 200% (p_{ratio} = 1). At equal strengths (p_{ratio} = 0) the threat is considered to be medium. The three fuzzy sets in the reputation membership function are defined to represent the diplomatic reputation of enemy, so their trustworthiness can be included in the evaluation of probability for their aggression. Performing the fuzzification of values p_{ratio} = 0.66 and reputation = -5 with the membership functions in Figure 20 results with the following membership degrees:

Table 6. Output membership degrees of the fuzzification.

Fuzzy Set	**Degree of Membership**
Weaker	0
Equal	0.34
Stronger	0.66
Bad	0.25
Average	0.75
Good	0

The get the threat level from these values, the following fuzzy rule matrix is used to map the values to Low, Medium and High threat levels:

Table 7. Fuzzy rule matrix for combining the force size ratio and reputation.

	Weaker	**Equal**	**Stronger**
Bad	Medium	High	High
Average	Low	Medium	High
Good	Low	Low	Medium

These above rules can be written as the following logic expressions:

```
mLow = (mWeaker ∧ mAverage) ∨ (mWeaker ∧ mGood) ∨ (mEqual ∧ mGood)
mMedium = (mWeaker ∧ mBad) ∨ (mEqual ∧ mAverage) ∨ (mStronger ∧ mGood)
mHigh = (mEqual ∧ mBad) ∨ (mStronger∧ mBad) ∨ (mStronger ∧ mAverage)
```

Populating the above expressions with the previously calculated membership degrees for the fuzzy sets results in the following values for the threat membership:

```
mLow = max(min(0, 0.75), min(0, 0), min(0.34, 0)) = 0
mMedium = max(min(0, 0.25), min(0.34, 0.75), min(0.66, 0)) = 0.34
mHigh = max(min(0.34, 0.25), min(0.66, 0.25), min(0.66, 0.75)) = 0.66
```

When the highest membership selection is used to determine the threat level, it can be concluded that mHigh has the highest membership value of 0.66, meaning that the threat level is high and decision making can act on strategic and diplomatic level accordingly. Another option would be using the membership blending method to calculate a numeric threat level value out of the membership values if needed to for example adjust internal diplomatic stance level.

Other applications

Besides the threat assessment, other applications for Fuzzy Logic in strategy games also include Bayesian Network probability reasoning [6 pp. 253-254] and decision making in Rule-Based Systems [17 p. 354]. Fuzzy State Machines can use Fuzzy Logic to do blending between states, allowing smooth transitions based on the Fuzzy transition conditions [17 pp. 364-369]. This technique is sometimes used for example to do animation state blending like in Unity's Mecanim. Other gaming genres, such as racing, can also benefit from the way Fuzzy Logic can be used to control vehicle steering, but that is out of the scope of this thesis [6 pp. 205-207].

3.4.4 Rule-Based AI

Rule-based Systems, sometimes also known as Expert Systems, have existed in the AI field since 1970s. They are sometimes considered a double-edged sword, as although they allow the experts to share their knowledge of the situation and reasoning about how to handle it, there is the downside that rule sets to define this knowledge must be created by those experts. Although rule-based systems have many applications outside gaming industry, such as in financial, medical and industrial software, there is also use for this approach also in games [4 p. 134]. Their strengths include the ability to use extensive rule sets to capture high-level knowledge of various complex problems [4 pp. 139-140], and the capability to make decisions in unexpected situations which cannot be easily handled with more simple approaches such as decision trees [17 p. 403]. The Figure 21 below shows the basic structure of rule-based system.

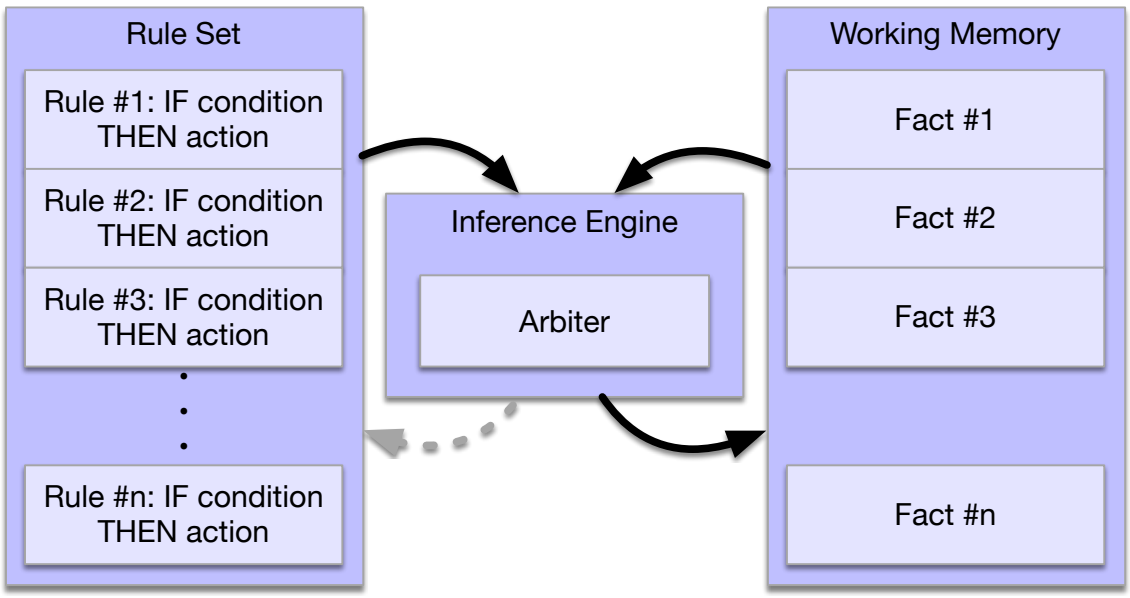


Figure 21. Overview of Rule-Based System [4 p. 138; 17 p. 404].

As pictured, this system consists of main components called the Rule Set, Inference Engine and Working Memory. Each of these parts are described below in more detail.

Rule Set

The actual knowledge of a problem is encoded in various rules, which are kept in the Rule Set, also known as the Knowledge Base. Each rule has two parts, the condition which must be satisfied for the rule to be fired, and action which defines what happens

when the rule is triggered. The condition can evaluate facts in Working Memory in various ways, and rules can also be enabled and disabled when needed. The action can alter facts in Working Memory, but can also control which rules are active, and even stop the processing completely [4 pp. 134-135].

Working Memory

All facts known by the Rule-Based System are kept in the Working Memory, which works as a database for the Inference Engine. Although the format of facts is not limited, they are usually stored as Boolean, numeric, string, and enumeration values [4 pp. 134-135].

Inference Engine

The actual processing of rules happens in the Inference Engine, which checks the rule conditions of the rule set and either selects the first match or uses the arbiter to choose which action to trigger. The processing takes place in iterations which continue until either no more facts are changed in the database, or a stop action is encountered. Usually forward chaining rule matching is used, but sometimes backward chaining can be used, which matches the rules based on their outcome (effect of actions on facts) instead of conditions, trying to find a starting state that can derive the expected result. [17 pp. 407-408] Rule-Based Systems are however notorious for suffering performance issues when large rule sets are used, which need considerable processing time. There are some optimizations for this process, the Rete Algorithm being one of the best known of them [4 pp. 134-135; 17 pp. 422-423].

Arbiter

Sometimes the system contains a separate arbiter component, which decides which rule is triggered if multiple rules are matched simultaneously during one iteration. Possible common approaches include using first applicable rule, least recently used rule, random rule, most specific conditions and dynamic priority arbitration [17 pp. 418-419].

Use case: Inferring tech tree state through rule-based reasoning

As rule-based systems allow inferring new facts from existing ones through the rules, one possible use for them in 4X strategy game is the capability for AI to use knowledge

about enemy's possession of a single technology to deduce the state of other technologies in the tech tree. This use case has been adapted and refined from the example provided by Bourg and Seemann [6 pp. 214-218]. The Figure 22 below shows a possible subset of tech tree.

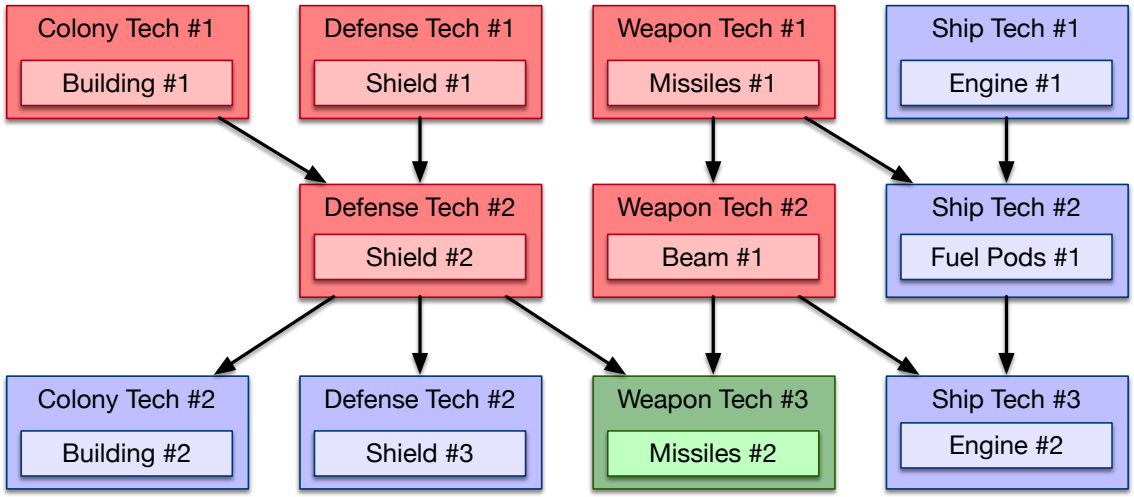


Figure 22. Inferring state of tech tree from knowledge of a single technology.

In the tree, technologies are connected by arrows leading from prerequisite technologies on higher levels down to the subsequent technologies on the next level. The following rules can be generated from this tech tree:

```
IF defense_tech_2 THEN colony_tech_1=Yes AND defense_tech_1=Yes
IF weapon_tech_2 THEN weapon_tech_1=Yes
IF ship_tech_2 THEN weapon_tech_1=Yes AND ship_tech_1=Yes
IF colony_tech_2 THEN defense_tech_2=Yes
IF defense_tech_2 THEN defense_tech_2=Yes
IF weapon_tech_3 THEN defense_tech_2=Yes AND weapon_tech_2=Yes
IF ship_tech_3 THEN weapon_tech_2=Yes AND ship_tech_2=Yes
```

As the AI has learned that enemy has a ship equipped with Missiles #2 upgrade (shown green in Figure 22), and thus has researched the Weapon Tech #3 technology, it can use the above rules to infer which other technologies are consequently possessed by the player as prerequisites. This starts by putting the fact *weapon_tech_3* into the Working Memory, and running the first iteration in Inference Engine, which finds a match for the following rule:

```
IF weapon_tech_3 THEN defense_tech_2=Yes AND weapon_tech_2=Yes
```

This rule adds the facts *defense_tech_2* and *weapon_tech_2* into the Working Memory. The inference engine runs next iteration to evaluate the rules, and continue the process

until no more new facts are added to the Knowledge Base, at which point the processing has finished. In Figure 22, the technologies set to "*Yes*" during this process are indicated in red.

Other potential uses in strategy games could be predefining certain events for scenarios (triggers) and possibility to control unit and strategic behavior with AI scripts implemented by AI programmer and/or designer.

3.4.5 Utility Theory

The idea behind the Utility theory has existed for a long time, and has a history predating even computers in the economics field where it is used to study consumer behavior and choices. The core concept in Utility Theory is scoring each action or state in the utility model with a uniform value, which represents the usefulness of each choice in the given context. To allow scores of multiple sources to be comparable, the utility values are normalized using methods appropriate to the given input data, and the scores can be combined from multiple sources to end up with final utility score which can be used to select the appropriate action. A simplified overview of the information flow inside a Utility System is shown below in Figure 23 [25].

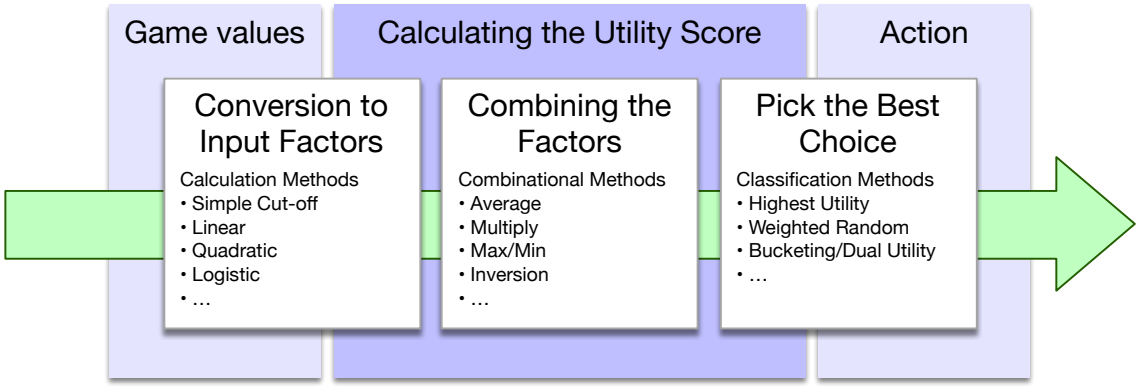


Figure 23. Simplified flow of information in a Utility System.

This flow of information inside Utility System can be roughly divided into the following phases:

Phase 1: Converting Game Values into Utility Factors

There is no hard-defined way of converting data into Utility Factors; the only rule is that all factors must have the same scale, so that they can be combined together and be comparable with each other. There are certain generally used methods for converting arbitrary game values, shown below in Figure 24. Other methods may also be used depending on what is required by the use-cases of the utility factors [25].

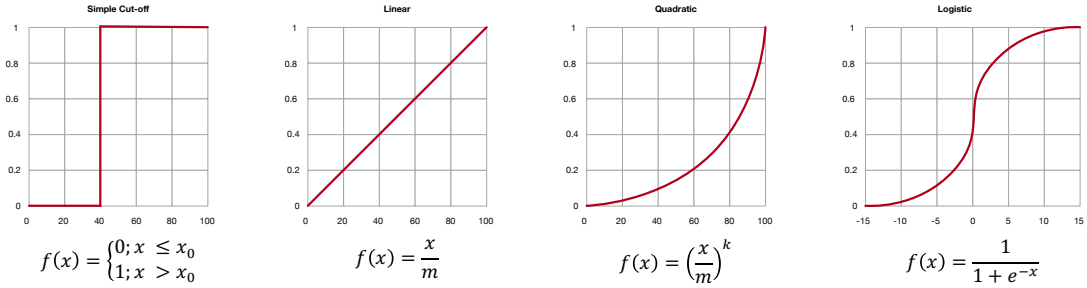


Figure 24. Commonly used formulas for calculating utility factors [25].

Phase 2: Combining Utility Factors

Usually there are more than one factor affecting the desirability of actions, and to get final utility scores for each of them they are combined. Commonly used methods include calculating average of utility factors, multiplying them together, picking the smaller or larger of them, or reversing the factor by inverting it. These operations can also be chained after each other, and exposing them as a visual graph editable by designers can be a very powerful tool. [25]

Phase 3: Picking the Best Action

After each one of the actions have been given a final utility score the AI selects which of them it should execute. The most straightforward way is to just pick the one with greatest utility, but it might in some cases lead to repeatable or predictable AI behavior. This can be overcome in some cases by using weighed random approach, where a random selection is made from possible actions with the utility score used as weight to give the actions with higher utility score a better chance of getting picked. This can also be combined with bucketing, also known as Dual Utility AI, in which the actions are categorized and assigned a bucket based on the effect they have. For example, when choosing production goal in a colony, the military units could be assigned in one bucket and colony

improvements in another one. With this approach, when building army has highest utility, it guarantees that a military unit is produced, but the type of unit can be randomized [25].

The utility scoring has a strong resemblance to fuzzification in fuzzy logic, and they share some principles especially in the way game values are converted into the internal representation in both approaches. They are also both good for promotion emergent behavior in AI when used properly by the AI designer [25].

Use Case: Diplomatic Decision Making

In the 4X strategy games, one possible use case for utility reasoning might be choosing an action during a diplomatic negotiation with another player. A simplified case for this is outlined below in Figure 25:

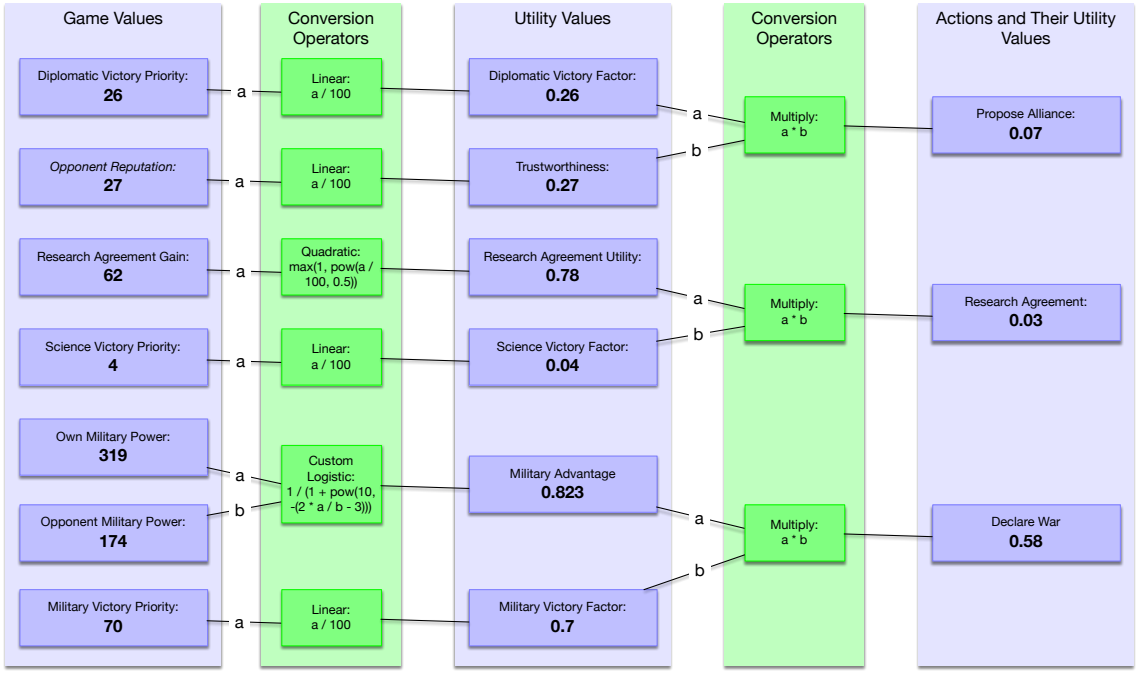


Figure 25. A possible Utility System for diplomatic decision making.

This situation assumes that the players have currently signed a peace treaty, which offers three possible actions: proposing alliance, signing a research agreement, or declaring war. There are also certain game values exposed to the Utility System: opponent reputation, score for scientific benefit of research agreement, and military strengths of own and opponent armies. Also, the AI personality goals are exposed as diplomatic victory, science victory and military victory priority values. There values are then converted

to Utility Factors using operators defined by the designer or AI programmer, which end up as utility values of the possible output actions as shown in Figure 25. This gives each of the actions a utility score, and if picking the action based on highest utility, the choice in this case would be declaring war against the opponent.

It should be noted that in an actual production implementation various other factors should be considered, such as how much the player likes or dislikes the opponent when declaring war and what impact it would have on the player's own reputation, how likely the opponent is to enter an alliance or research agreement before offering them, and many others.

Using utility scoring for decision making is especially fit for the type of strategy games that this project represents, as due to the nearly infinite number of possible moves per turn there is no way to score individual actions in a purely deterministic way. In this situation, the reasoning of utility of actions allows the AI to make educated "best-guess" choices based on the available data of the game state [25]. The utility-based approach is also highly versatile thanks to its simple concept, which helps it combined with a number of other techniques such as implementing utility selector in behavior trees [24] and applying utility-based costs in colony production goal trees [26].

3.5 Influence Maps

To allow strategic analysis of map and game world, various types of influence maps can be utilized to give the AI environmental awareness. Some examples of use cases for this data are listed below [27]:

- Pathfinding can include influence as part of heuristic to avoid or favor certain areas
- Weak spots in enemy influence can be used to target attacks in planning of higher-level military operations, or to prioritize reinforcing own territory.

The basic structure and function of influence maps has similarities to cellular automata, in which uniform grids of values are modified by certain rules as a function of time, usually based on the values of surrounding tiles. One classic example is Conway's "Game of Life" which uses a very simple set of rules, although cellular automata has many other

higher-level uses such as city simulation in SimCity [17 pp. 536-537]. The data in influence maps can be composed of multiple layers of data, including for example [17 pp. 499-512]:

- Tactical Analysis
    - Friendly and enemy unit and point-of-interest threat generation
- Terrain Analysis
    - Defensive and/or movement bonuses from terrain
    - Map visibility, which can be used to either increase the “threat of unknown” or to prioritize exploration
- Learning
    - Past events recorded on map, such as unit kills, i.e. “frag map”

Some of the data, such as terrain analysis, is by default spread on the influence map layer uniformly, and can be used as input as such. Some other data though, like tactical positions such as unit threat, are localized to single spots in the map, and their influence needs to be distributed on the layer to be usable. To do this, there are a couple of common options available shown in Table 8:

Table 8. List of common influence calculation methods [17 pp. 502-505].

Method	**Description**
Limited Radius of Effect	Influence is applied on map as a function of distance to the unit, with fixed falloff.
Convolution Filters	The unit influence is applied on map using two- or three-dimensional filter matrix, for example using Gaussian blur.
Map Flooding	The unit influence is propagated on map using Dijkstra or A* algorithm.

It is also possible to use variations of the above methods, depending on the source data and how the influence map is used in the game. This involves usually fine-tuning by the AI programmers and designers to find a good balance for the influence which benefits the AI in decision making. For example, if certain areas of map are not visible to the player, it is good idea to take the factor of unknown into account when calculating influence; this however means that each AI player needs to run its own analysis of the influence map, in contrast to a game state where all players have the same knowledge of

unit positions and strengths, in which case the data could be shared [17 pp. 505-507]. An example of a simple influence map is visualized in Figure 26.

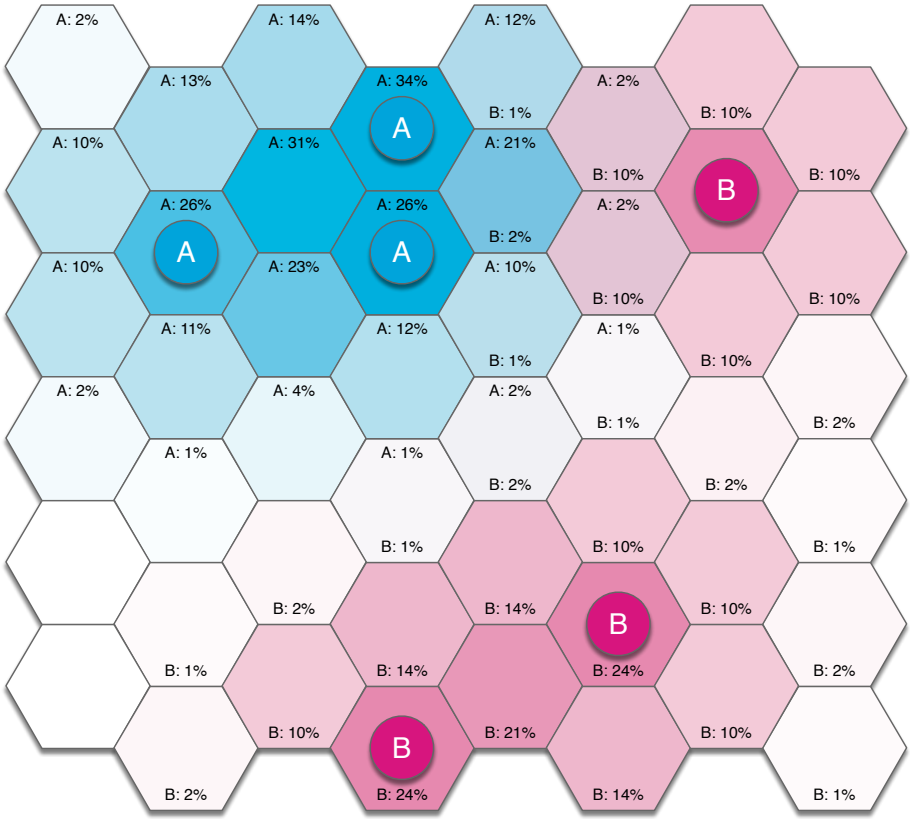


Figure 26. An example of map of unit threat influence on hexagonal grid.

This example shows a hexagonal grid with three units belonging to each player A and B of equal influence value. In this case, the unit influence values were propagated on the map using normalized Gaussian blur convolution filter applied through a rank 3 tensor on cubic hex projection plane (q + r + s = 0).

Spatial Database

One possible way to represent the different sources of data affecting influence is the use of spatial database, as suggested by Paul Tozour. In his approach, the data is applied to distinct layers in a generalized way, with some possible examples of data layers listed below [28]:

- Openness layer
- Cover layer
- Area searching layer

- Line-of-fire layer
- Light level layer

The layers can be combined using various algorithms at runtime, for example using a formula like the one in Equation (6) to calculate dynamic desirability layer from other source layers [28]:

$$desirability = openness \times occupancy \times static\_cover \qquad (6)$$

One of the possible benefits of using this layering of data is the increased tendency of emergent behavior in AI unit coordination through the use of shared data structures [28].

Strategic Dispositions

When units are being categorized in order to identify strategic dispositions, the information in spatial database can be used to aid this purpose. The knowledge can be used in tactical analysis and decision making, for example to identify weak spots which can be engaged in enemy territory, or areas in own defences that need to be reinforced [29]. An example of evaluation of strategic dispositions is shown below in Figure 27.

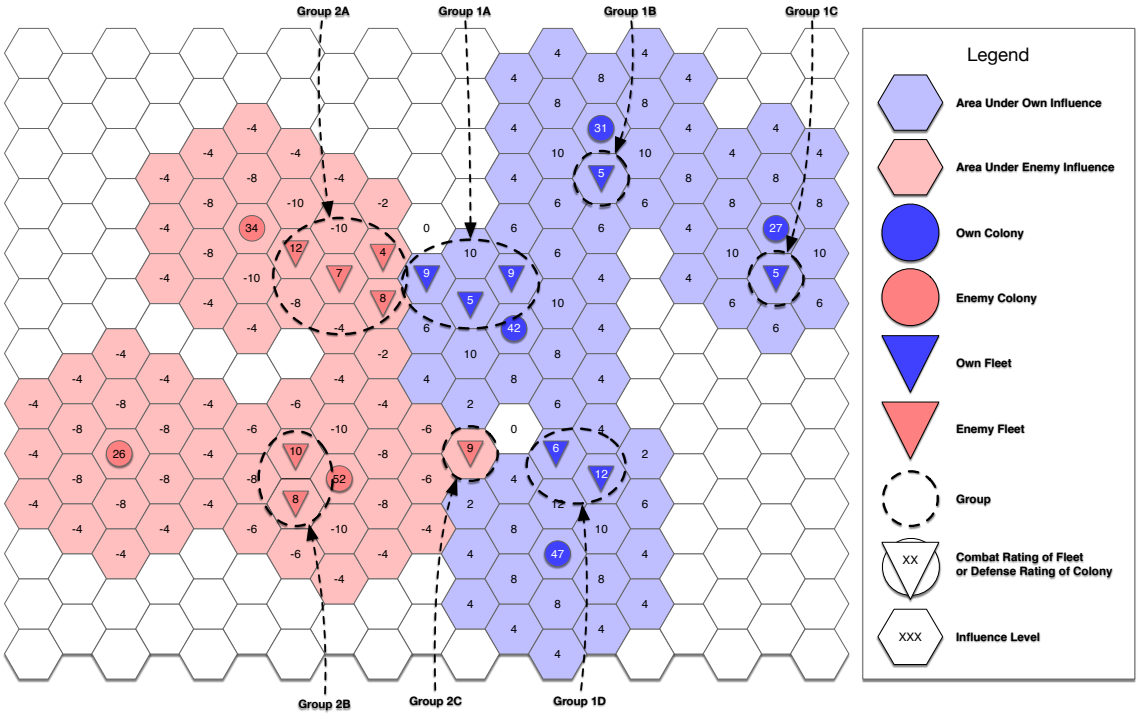


Figure 27. A possible grouping of units for analyzing strategic dispositions.

Figure 27 shows a case where a simplified map of fleet and colony influences has been propagated on the map using limited radius with fixed fall-off, and clusters of unit have been grouped using a simple algorithm which selects the strongest units and units in their immediate vicinity to be part of their group. The total strengths of each group of units is known, and thus their threat level can be estimated using fuzzy logic methods similar to the ones demonstrated earlier in Chapter 3.4.3. This data can be combined with the influence map, for example by calculating the gradient of influence level between nearby grouped units. In this case, a higher gradient would indicate higher tension between units, which can be used as input data to the tactical analysis algorithm which directs the units in groups to either engage enemy unit groups, or to reinforce the defenses on local territory.

The actual implementation of selection of actions depends on the iterative experimentation by designers and the AI programmer, but could for example use utility-based scoring based on the input factors gained from the analysis.

Tactical Pathfinding

The influence maps can also be used in pathfinding to allow the units to consider possible threats when planning the route to the target position. The tactical pathfinding can be implemented easily by adjusting the heuristic function of the A* pathfinding algorithm for example by adding penalty based on enemy threat level on the influence map, which makes the units evade dangerous areas of the map, giving the AI movement choices are stronger impression of intelligence. One challenge in this approach however is the care needed when applying changes to the scoring heuristic function to avoid increasing the cost of pathfinding processing time too much [30].

A possible use case for this approach in a 4X strategy game might be for example the need to plan route for worker unit across unclaimed space with recently observed enemy movement. In this case, the pathfinding should avoid areas which would most likely to lead to encounter with enemy.

3.6 Goal-Oriented Behaviors

With the previously described methods, it is possible to build an AI that can evaluate the current game state and choose appropriate actions which appears sufficiently intelligent in casual gameplay. However, it is especially important in strategy games for the AI to be have long-term strategy and goals which makes AI's actions and decisions more meaningful, and thus giving more challenging and meaningful gameplay experience for the players. To accomplish this, various forms of Goal-Oriented Behaviors (GOBs) can be implemented which can give the AI not only immediate internal needs which it aims to fulfill, but also the capability of chaining multiple actions together in order to reach more complex goals [17 pp. 376-377].

This section introduces a few key technologies that can be used to implement this kind of behavior which is useful in the higher layers of the multi-tier AI mode, including production and research planning, which involves also coordination between different higher-level agents.

3.6.1 Goal-Oriented Action Planning

The idea behind Goal-Oriented Action Planning (GOAP) has long history, having roots in the Stanford Research Institute Problem Solver (STRIPS) which was created already as early as in the 1970s [21]. GOAP planning uses backward-chaining search, which means that it uses the desired goal state as starting point, and traces the action sequence which leads to the starting state. There are a few basic building blocks in this approach [31]:

Goal

A goal represents the desired final state which the planner should attempt to reach. Each goal has a set of conditions which must be satisfied for the goal to be reachable.

Action

There is a predefined set of actions, each of which represent what the AI can do. Each action has set of preconditions and effects; the preconditions define what the world state

should be for the action be doable, and the effects define how the world state is changed by this action.

Plan

The final plan is a sequence of actions leading from the current world state to the desired goal state.

World State

The GOAP planner uses symbolic representation of world to perform search in state-space. This abstraction allows both preconditions to be matched against the world state, and effects can also be used to apply changes to the simulated states.

Planning process: The simplistic approach

When running plan formulation, the GOAP planner is given the desired goal state, a list of possible actions, and the current world state which is abstracted from the concrete game world into the symbolic presentation. The Figure 28 below shows a simplified overview of the planning process in state-space during the plan formulation.

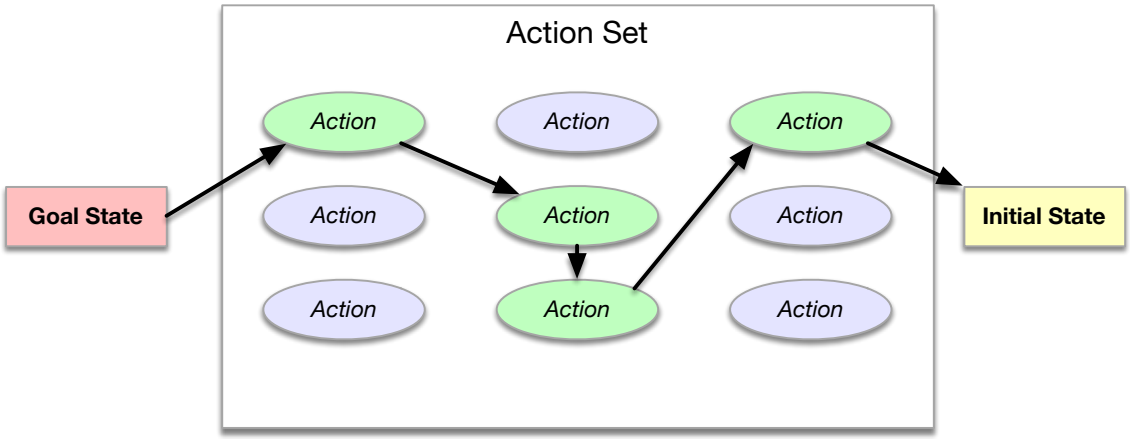


Figure 28. Abstract illustration of GOAP planning process.

The planner starts from the desired goal state, adding its conditions into the list of unsatisfied world properties. During each iteration, the planner searches for actions which have effects that match the unsatisfied world properties. Each of the possible actions is picked as a possible node in the search graph, and evaluated recursively by applying

the effects to world state and adding the preconditions of the action to the list of unsatisfied world properties. When the planner reaches a state where all the world properties are satisfied, it has reached the initial state and thus has found a valid plan, or if no more actions can be matched in which case there is no solution. After a valid plan has been found, it is made active and the AI attempts to follow it. However, if any alterations are made to the world state during the plan execution, replanning is required and thus the planning process is run again [31].

Adding action costs to the plan formulation

Sometimes just knowing a possible sequence of actions for reaching the goal does not suffice, as there might be other lower-cost paths leading to it. The types of cost factors depend on the use case, for example time, money or health.

When cost is added to the actions, the search space can be considered as a weighed graph which can be evaluated using A* algorithm with the expected cost used as heuristic for the search formula. Figure 29 below shows part of a possible search tree which might be formed during GOAP planning.

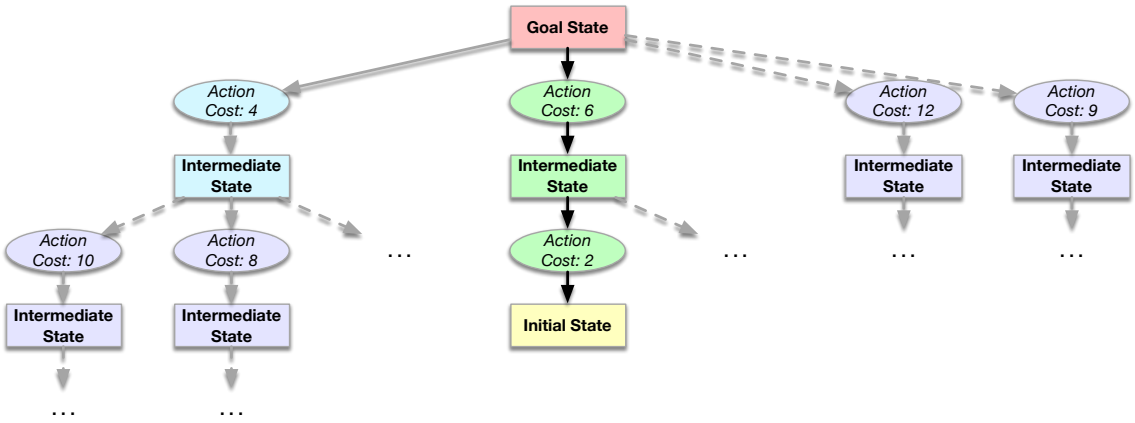


Figure 29. A partial state-space search tree for GOAP.

A benefit of the state-space presentation is that as it is practically a game tree structure, certain traditional board game AI techniques can be applied to it. For example, to prevent unnecessary time spent on evaluating duplicate subtrees, the evaluated states can be stored in a transposition table, with the hash of symbolic world state used as cache key. Other benefits include the possibility of using Alpha-Beta Pruning and the Killer Heuristic.

Iterative Deepening A* (IDA*)

When A* is used for pathfinding, each graph node is only evaluated once and there is limited number of nodes to explore. However, with GOAP planning there is no limit on how many times a single action may be performed, leading to infinitely long action sequences. To avoid this, Iterative Deepening A* (IDA*), which is a variant of Iterative Deepening Search (IDS) algorithm can be used for traversing the state graph [17 pp. 376-401]. The progress of IDA* search is shown below in Figure 30.

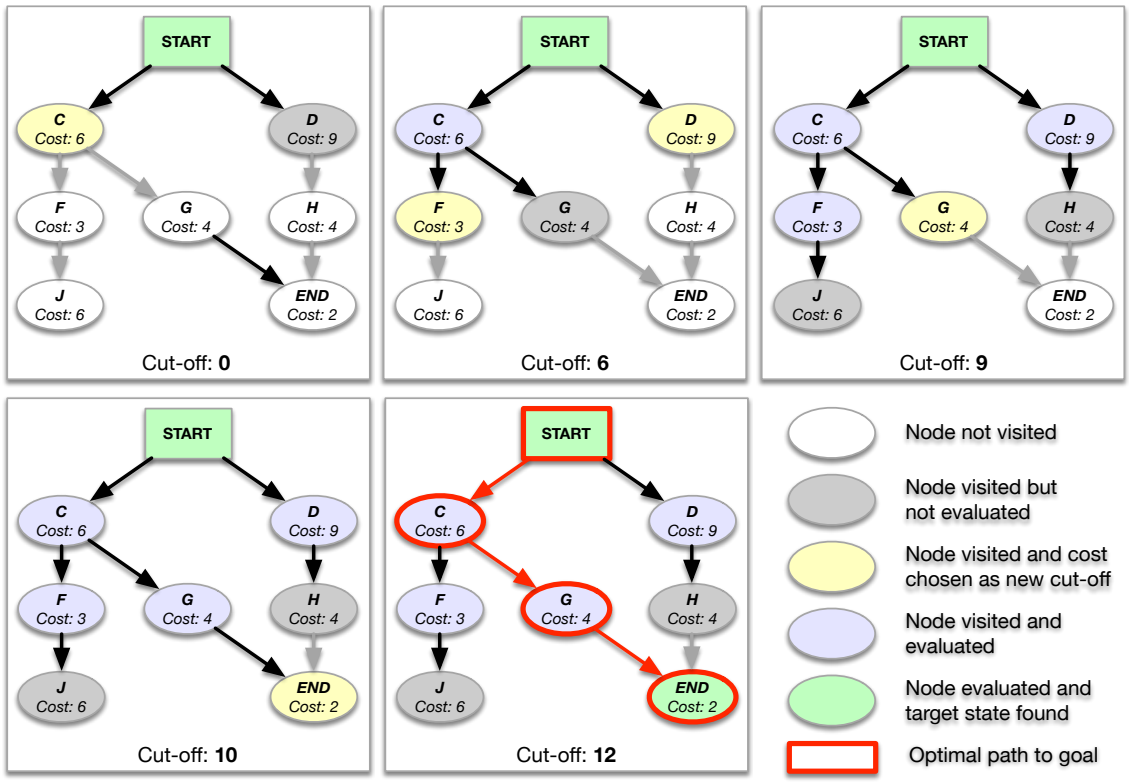


Figure 30. Iterative Deepening A* search example.

The IDA* search works by defining a cut-off value, which is the maximum cost until which the search iteration terminates. On each iteration, the regular depth-first search is run until the current cut-off limit is reached. If the target node was not found, the cut-off value is increased and search is run again, thus iteratively progressing further in the search tree each time [17 pp. 376-401]. The above Figure 30 shows roughly how this iterative progress works.

3.6.2 Hierarchical Task Networks

Although sharing some concepts with STRIPS planning, the Hierarchical Task Network (HTN) approach assigns the current world state as starting point, and uses available tasks to construct the plan through forward-chaining task decomposition. This is opposite to previously introduced GOAP, which uses backward-chaining search in the state-space graph to find plan leading from goal state to the initial world state [32]. The Figure 31 below shows an overview of a simple HTN planning system adapted for games.

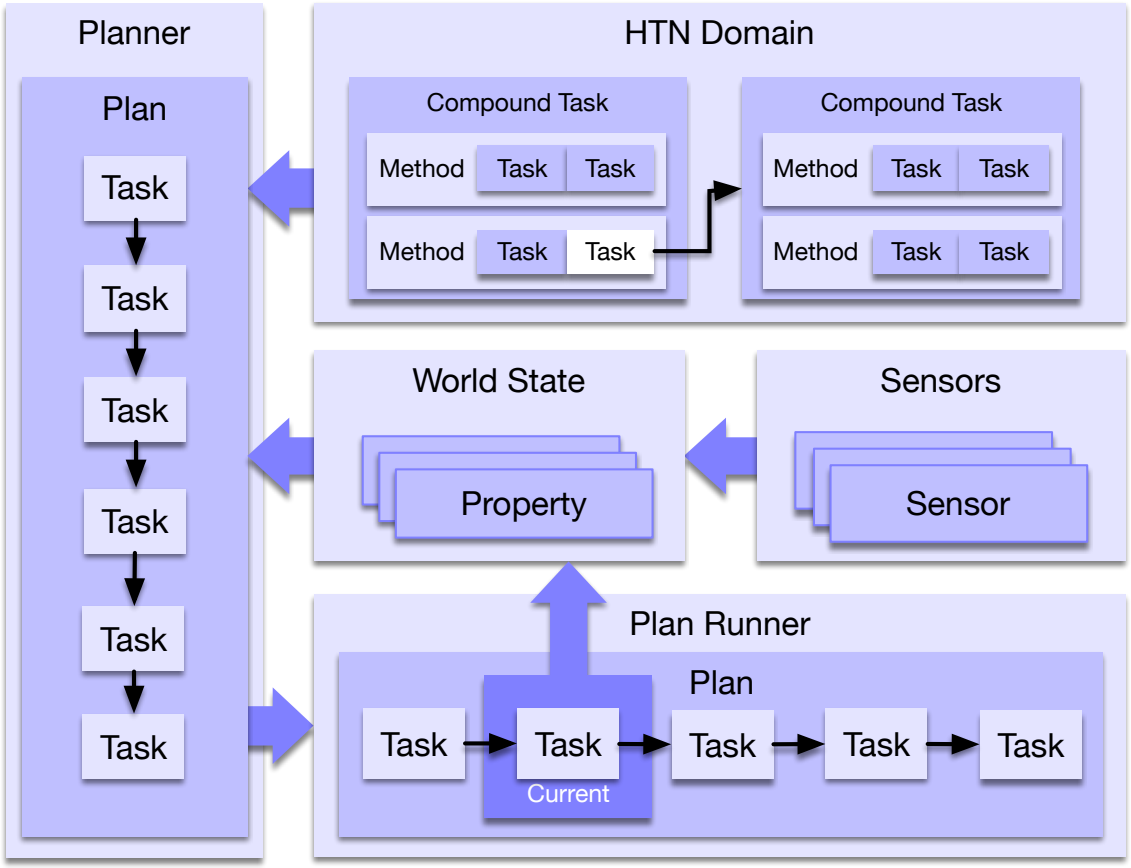


Figure 31. Overview of HTN planning system [32].

This HTN planning system is divided into the following components:

HTN Domain

The essential part of HTN planning system is the HTN domain, which contains all tasks available for solving the particular problem. There are two main types of tasks:

- **Primitive Tasks**, which are the basic building blocks of the plan. They contain an operator which defines the actual low-level task for the game, condition which uses the world state properties to evaluate whether the task can be executed, and effects which alter the planner world state.
- **Compound Tasks**, which contain multiple methods of executing a particular task. Each of these methods have set of preconditions that dictate which of the methods (if any) gets chosen to be decomposed into the plan based of the current world state.

The tasks available in the domain form a hierarchy, hence giving the name for this planning approach.

World State and Sensors

Like in the GOAP approach, the planner uses an internal world state to simulate effects of tasks in the game, using the resulting state in primitive task conditions and compound task preconditions to control the planning process. Sensors work as adapters providing the simulated world state from actual game state.

Planner

The planner does the actual planning work, which starts from the root task in the HTN domain, which gets inserted into the list of tasks to process in beginning of this process, after which the iterative planning process is started. On each iteration, the first item in the list of tasks to progress is dequeued and processed. If the item in list is a primitive task, its condition gets run, and if satisfied, the task gets appended to the final plan. If the task is a compound task, the preconditions of the methods get run to select the appropriate method to decompose. This decomposition enqueues the tasks in the method in front of the list of tasks to progress. The iterations continue until the list of tasks to progress is empty. An example planning case is shown below in Figure 32.

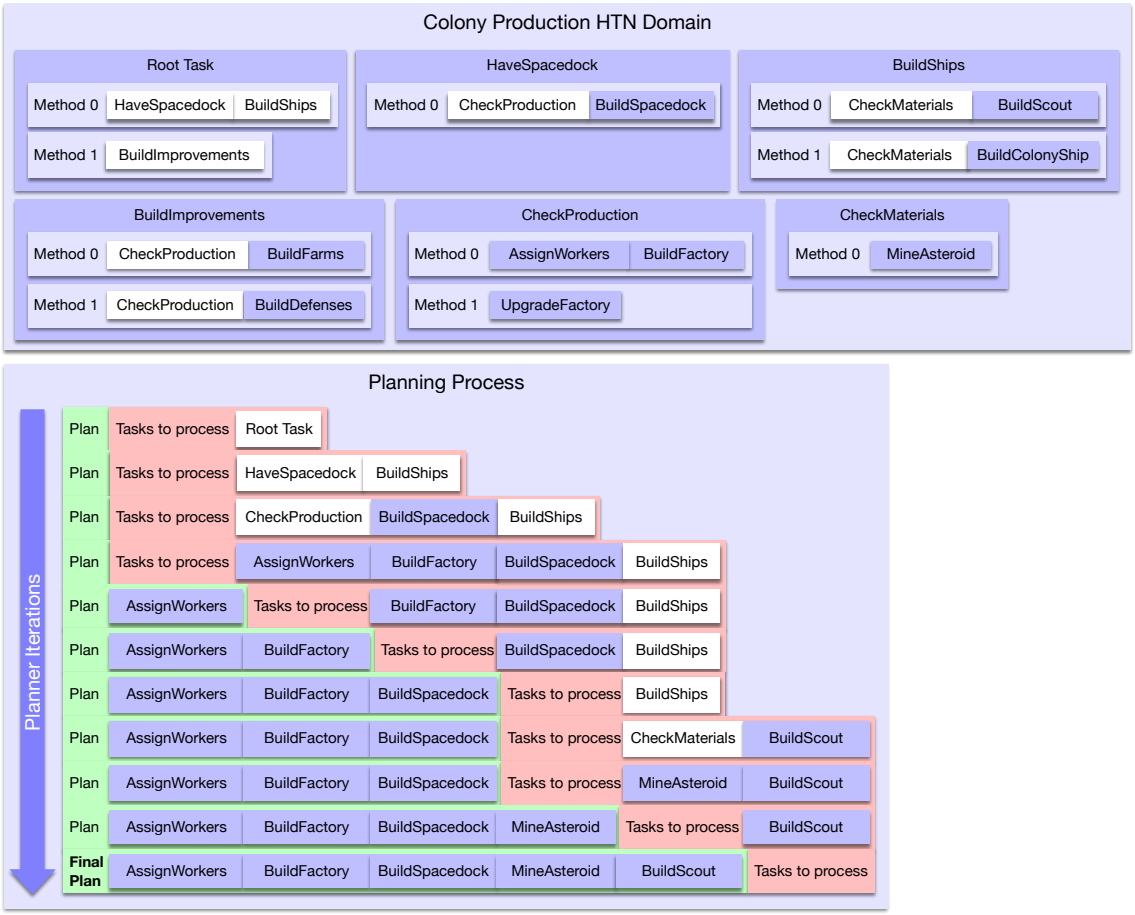


Figure 32. Illustration of a simplified HTN planning example.

After the planning process is completed, the planner has the final plan which can be passed to the plan runner. Depending on the structure of the HTN domain, it is possible that the planning may in some cases fail to provide any valid plan at all.

Plan Runner

After the planner has created a plan, the plan runner starts executing the plan during gameplay, keeping track of currently active task in the plan and checking the task conditions during this process. If the world state gets changed unexpectedly during plan execution, i.e. when the state does not match the conditions of task executed next in the plan, the HTN is forced to do a replan to run the planning process again.

3.6.3 Composite Tasks

One approach to handling goal-oriented behavior is the use of the Composite Task architecture. Originally implemented in 1995 for a real-time strategy game, it has since

found use in CSXII Tactical Combat Simulator used by the U.S. Army [33]. The Figure 33 below shows the basic structure of Composite Tasks.

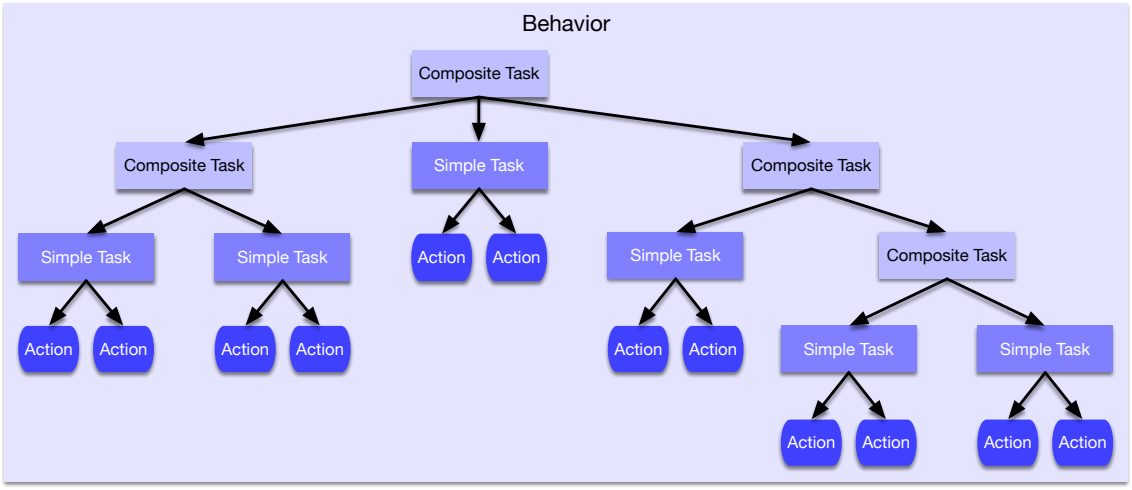


Figure 33. Structure of Composite Tasks.

The main concept in Composite Task model is the ability to split a high-level main goal into smaller subgoals, creating a hierarchy of tasks. This flexibility allows expressing even very complex goals and how to satisfy them using combination of low-level actions. The Composite Tasks consist of the following components:

- **Composite Tasks**, which can contain either other Composite Tasks or Simple Tasks
- **Simple Tasks**, which contain one or more Actions
- **Actions**, which are individual atomic operations that the AI can perform

The execution of tasks starts from the root task, which evaluates its child components in priority (usually left-to-right) order, until the entire hierarchy has been walked through. The benefits of Composite Tasks include design simplicity, data-driven content and generalized evaluation process [33].

3.6.4 Multi-Unit Planning with Hierarchical Plan-Spaces

The traditional planning methods are well suited for planning actions for a single actor, when the number of possible actions stays in reasonable amount. However, when planning actions for multiple units at once, for example for military incursions, the state-space searching suffers from combinatory explosion. This means that the number of possible

combinations of actions grows exponentially exceeding the available processing power and thus becoming unusable. To solve this problem, the planning can be done in plan-space instead of state-space [34]. The Figure 34 below shows abstract illustration of the difference between state-space and plan-space planning.

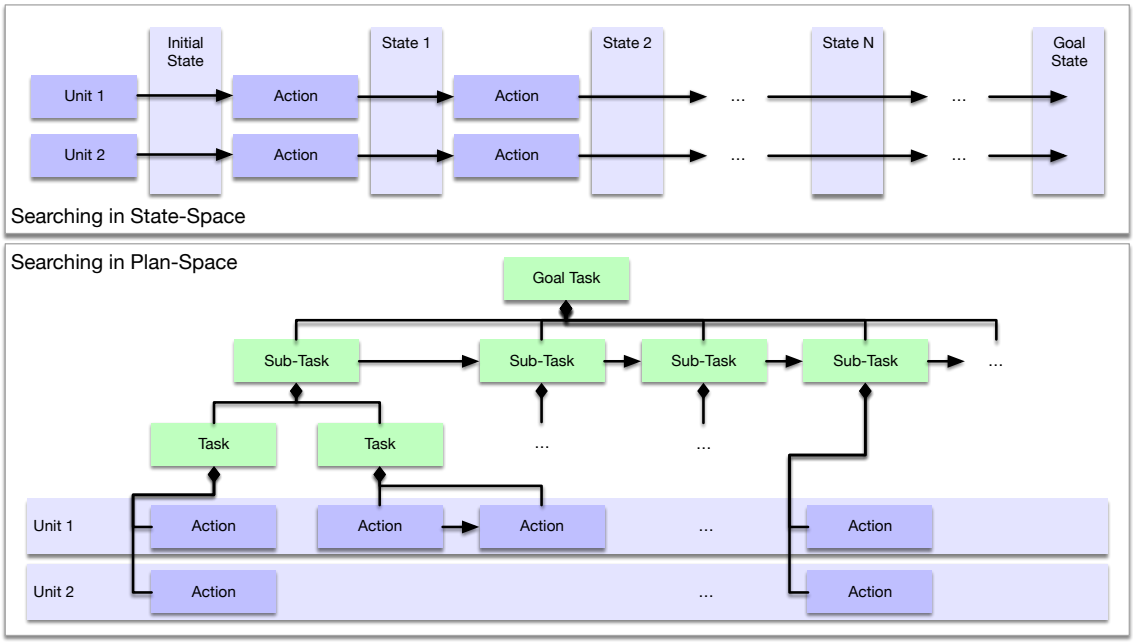


Figure 34. Comparison of state-space and plan-space planning [34].

With this approach, the planning is started from high-level task, which is further refined into lower-level tasks, and eventually individual unit actions [34].

The planner

Instead of keeping track of possible states, the planner uses list of possible plans and scores them based on their expected cost, using the same A* algorithm like in other graph-based planners. On each iteration, planner dequeues the most promising plan (i.e. the one with lowest estimated cost), select the appropriate planner methods and their alternative approaches to get a list of new possible plans to branch off from this plan. Each of these new plans get refined by the planner method, after which their estimated cost is calculated by the task cost estimation function, and they are queued into the list of possible open plans. This iterative process runs until either a planner succeeds by finding a complete plan in the queue, or fails by running out of plans to refine [34].

The tasks

The plan is composed of a hierarchy of tasks that can either be compound tasks which can be further refined to other tasks, or primitive tasks, which represent individual unit actions. The planning domain is defined by list of possible tasks, which belong to specific scope in the domain based on their position in the task hierarchy. Some possible tasks are listed below in Table 9.

Table 9. Some possible tasks for the plan-space planning (adapted from [34]).

Scope	**Task examples**
Mission	High-level mission task
Objective	Capture Colony, Defend Colony
Group	Form Up, Eliminate Colony Defenses, Attack Invaders
Tactic	Bombing Run
Units	Attack Fleet
Individual Unit	Move, Attack, Wait, Bomb, Defend, Deploy Troops

The individual tasks set of inputs and outputs, which are used to link unit states between sequential tasks, usually providing the output state of previous task as input of the next task. When tasks are chained sequentially, the preceding tasks are required to be completed before the next task in sequence can be activated. This allows the planner methods to control which tasks can be executed in parallel and sequential order.

Planner methods

The actual refining of tasks is done by planner methods, which take the current plan and task to be refined as input, and provide a refined plan for the planner. The planner methods only apply to specific tasks, and their complexity ranges from simple single task output to complex combination of tasks. They work by creating a number of subtasks for the task being refined, thus expanding the current plan to lower level. The planner methods matching the tasks in Table 9 are listed below in Table 10.

Table 10. Some possible planner methods (adapted from [34]).

Scope	Planner method examples and responsibilities
Mission	Allocate units
Objective	Define activities, assign units to groups
Group	Execute tasks as groups
Tactic	Synchronize tactical activity
Units	Arrange cooperation between units
Individual Unit	Define the actions

The plan-space graph

As mentioned earlier, the planner maintains a list of all possible complete and non-complete plans as it searches through the plan-space. A part of this plan-space graph is illustrated below in Figure 35:

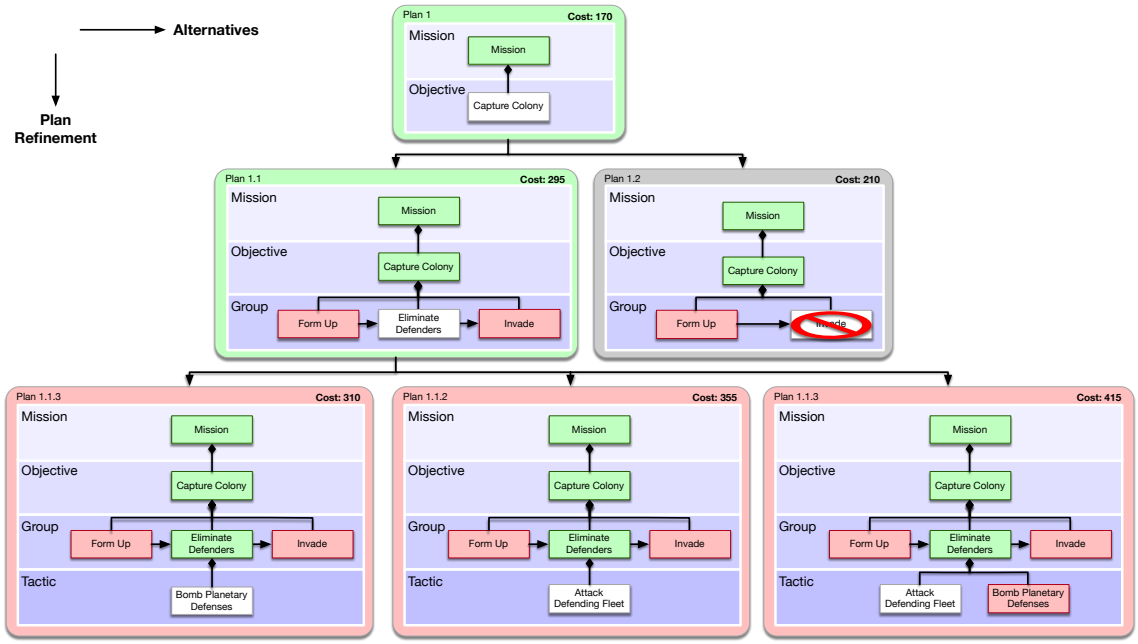


Figure 35. Illustration of the plan-space graph (adapted from [34]).

In the illustration, each plan is shown as a branch in the plan-space graph with their associated cost estimate. The green plans are in the closed-list of plans that have been refined, and red plans are in the queue of open plans. The tasks inside each plan show how deep the particular plan has been refined; green indicates tasks that has been refined, white shows the task being refined now, and red tasks are unrefined tasks.

Use case: Planning attack on enemy colony

The hierarchical plan-space planning has various uses in a 4X strategy game, and this example focuses on a simple case of attack on enemy colony. The player has four fleets available to be allocated for this mission: a battleship fleet, a destroyer fleet, a bomber fleet and group of troop transports. The resulting plan that can be generated using the example tasks listed previously in Table 9 is shown below in Figure 36:

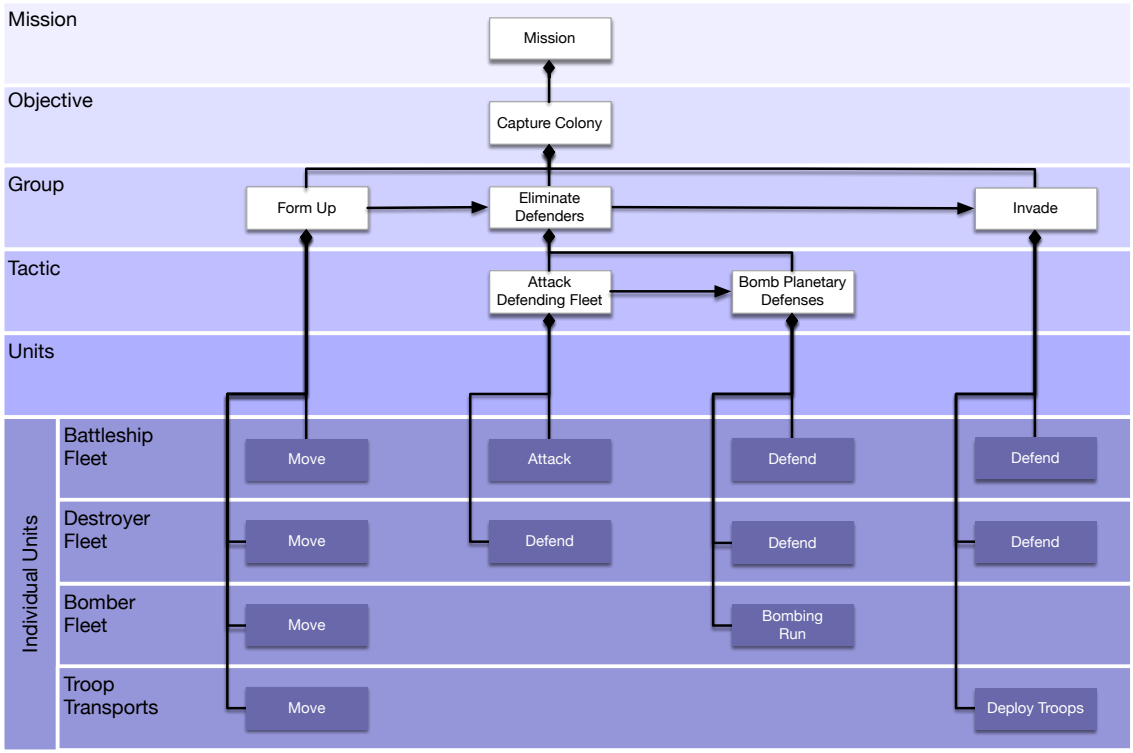


Figure 36. An example plan for invasion of enemy colony.

The planning starts by creating the capture colony task on objective layer, which contains information about the target colony, its defenses, and the AI player's available fleets for the mission. After this, the objective gets further refined into set of group-level tasks: Forming up the fleets, eliminating defenders, and invading the colony.

The form up task can take advantage of influence map and tactical pathfinding to pick the best positions for the fleets, and use this data to create the tasks for fleet movement to those positions. It can also consider the vulnerability of certain unit types, such as troop transports, when it assigns these positions.

The task for eliminating defenders can have various alternatives depending on the type and number of defenders in the colony; presence of enemy fleet creates the need for attacking enemy fleet, and existence of any planetary defenses requires using bombers to eliminate them when other fleets are protecting the bombing run. These tasks are further refined down into actions for the appropriate fleet types available.

The last task in capturing the colony is planetary invasion, which in straightforward way creates the troop deployment actions for troop transports and assigns the other fleets to defend the transports.

3.7 Diplomatic Reasoning

A game featuring players with the ability to engage in diplomatic relationships with each other imposes a certain set of requirements for the AI:

Forming the opinion of other players

The most important part of diplomatic interaction is the ability to evaluate opinions about other players; how they have behaved in the past, what they are expected to do, what things the players agree and disagree about, how the military, scientific, economic and social status are evaluated, etc. This includes the mechanism how the actions of players affect these opinions, and how these opinions end up shaping diplomatic relationships into friendships, alliances, enmity or animosity. A method for handling opinions is presented later in Chapter 3.7.1.

The ability to estimate percussions of actions

When it comes to making diplomatic decisions, the AI player needs to be able to understand effects of its actions. With the opinion system in place, the AI can use the expected opinion changes in goal tree search with combination of utility scoring to evaluate its actions, and choose the one which yields the highest score. With this approach, the AI can utilize knowledge of army sizes, the players' opinions about each other, and other factors such as personality weights in making educated guess for results of the actions.

Illusions of a character with personality behind the AI player

If all AI players would make decisions based on the same goals and using the same scoring methods, there would be little variation between the different types of players in the game, rendering the AI behavior more predictable and boring, and removing any differences in behavior among the opponents. With certain preset weights given to each of the AI players, their decisions can be influenced to be focused on unique goals, giving each of them a more distinct personality. This also allows a human player who wants to play the game with a certain strategy to seek alliances with AI players which have goals and priorities matching his/her own goals.

Long-term goals and persistence

The AI needs to have logical goals and the ability to make long-term decisions to help forming alliances and other agreements with other players. The combination of player-specific weights and opinion system create a natural foundation for this process.

3.7.1 Opinion Systems

The original approach to Opinion Systems as used by Adam Russell is focused on allowing individual NPC agents in game world to shape their opinion about other players based on their actions, but the core mechanism he proposed can be adapted for controlling the opinions of virtual players in AI about other players [35]. The Figure 37 below shows adaption of Russell's Opinion System for a 4X strategy game diplomacy.

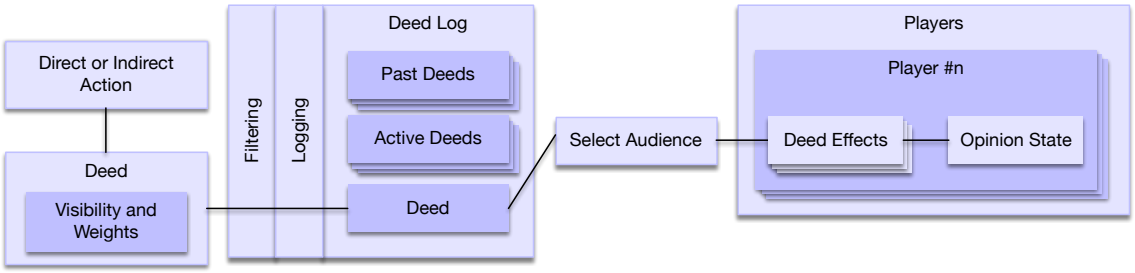


Figure 37. Opinion System adapted for 4X strategy games.

The main differences from his model are the replacement of NPC characters with virtual players, and replacement of global opinion with visibility and weights used to select audience of the deeds, and other minor adjustments to handle local opinion transformations.

Opinion State

The central piece of data in the Opinion System is an opinion state, which contains the opinion in either discrete or numeric format. This could for example be one player's trustworthiness opinion about another player, ranging from -1.0 to 1.0. These opinion states can be either simple single-track values, or multidimensional opinions which are composed of more than one value affecting the opinion state [35].

There are both positive and negative aspects of the multidimensional approach; the positive features include orthogonality, greater variety of effects, better match to natural language and having more information. The downsides however include being more brittle at design changes, increased confusion, difficulty in visualization and challenges in quantization [35]. Table 11 lists a subset of the possible opinion values that can be used in 4X strategy game diplomacy, with multidimensional approach involving four different opinion values.

Table 11. Examples of some potential opinion values in diplomacy.

Opinion Value	**Meaning of -1.0**	**Meaning of +1.0**
Scariness	Unthreatening	Terrifying
Trustworthiness	Deceitful	Honest
Sentiment	Loathed	Admirable
Aggression	Pacifist	Warmonger

This approach has the benefit of giving diplomacy more depth, for example if a player has strong army and thus high scariness score, but low sentiment score for past offenses, another player might be unwilling to enter into a trade pact even if they would fear the opponent's army [35].

Actions and Deeds

The deeds originate from either direct or indirect actions made by the player. Direct actions might include declaration of war or using spy to perform a sabotage mission, while massing troops could be induced as indirect action measured using threat analysis and influence maps. It is possible that one action causes multiple deeds, such as declaring war during peace treaty would also raise the break treaty deed. Some examples of possible deeds are listed in Table 12 below, with their associated audiences and weights.

Table 12. Some possible deeds and their weights.

Deed	**Affected Opinion**	**Audience**	**Weights**		
			Target	**Allies**	**All**
Declare War	Aggression	Public	+0.5	+0.5	+0.1
Break Treaty	Trustworthiness	Public	-0.5	-0.4	-0.2
Demand Tribute	Sentiment	Public	-0.2	-0.2	-0.2
Sabotage	Sentiment	Allies	-0.2	-0.1	
Mass Troops	Scariness	Private	+0.1		
Trespass Territory	Scariness	Visibility	+0.1		
Mass Genocide	Sentiment	Public	-0.5	-0.5	-0.5

There are four different audience types in this model:

- **Private**: Only target player receives the deed notification
- **Allies**: Target player and its allies receive the deed notification
- **Public**: All players receive the deed notification
- **Visibility**: Players who have currently visibility of the affected map square get notified

The deed audience type is specific to each deed type, and posting a deed requires either a target player or target location depending on the type.

Deed Log

The deeds are posted into Deed Log through filtering and logging pipeline, which is used to allow for example to temporarily suspend delivery of certain deeds, and to track statistics about the deed posts. The Deed Log not forwards the deeds to the subscribers of deed events, but also keeps track of past deeds, and has a list of persistent deeds (for example, trespassing enemy territory could be handled as a persistent deed, which posts the deed notification every turn until it gets deactivated when the units leave enemy territory) [35].

Audience Selection

The original audience type and the target passed with deed are used to pick the destination of the deed, and each of the recipients get notified.

Local Transformation

Before the deed is used to affect a player's opinion, it goes through local transformation which is specific to each deed type. In this modified 4X strategy game model, this transformation process is used to apply in certain cases multiplier to the deed weight based on the notification recipient's existing opinion about the target player. For example, let's assume that player A demands tribute from player B, incurring sentiment penalty for player B's opinion about player A. If there exists player C, which has negative opinion about player B, then player C would use negative multiplier for local transformation about the deed weight, leading to positive sentiment offset for player C's opinion about player A [35].

Deed Effects

When the deed notification eventually receives the player, it has to have effect on the receiving player's opinion. This change in opinion is invoked as transient offset through various possible offset functions, one of which is specific to each deed type. Example of a transient offset function is shown below in Figure 38.

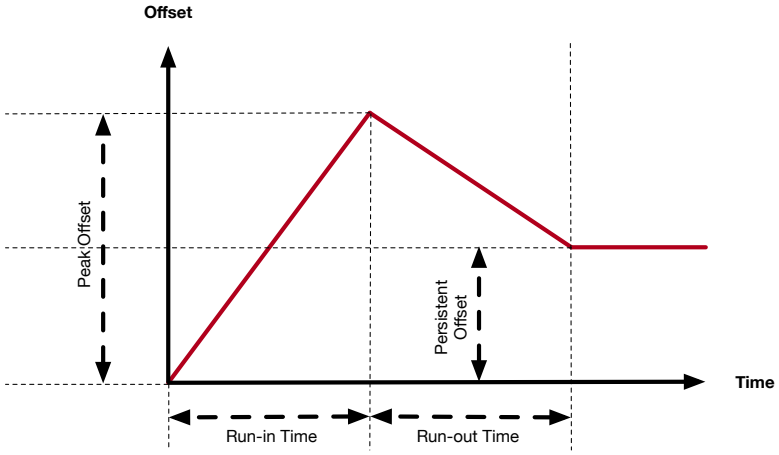


Figure 38. Opinion transient offset function example by Adam Russell [35 p. 544].

The function has a run-in time during which the deed offset increases until it reaches the peak offset and highest effect on the opinion. After this it decreases during the run-out time, and when the transient offset function ends the final offset is left as the permanent change in opinion. This allows for example a genocide deed to have strong permanent impact on opinion, but a trespassing of territory yields only a temporary change which dissipates gradually [35].

To prevent excess accumulation of transient offset effects on opinion by repeated deeds, the effect frequencies can be regulated by adding a minimum time between repeated deed effects for a single player [35].

3.8 Customizing AI

Although it is possible to implement AI completely by hard-coding it within the game code, there are motivations for making AI customizable which are twofold. For the first, the development of AI is collaboration between programmers and designers, and to support this process the designers should be able to project their visions in the game with as little friction and delay as possible. And for the second, by providing flexible methods of modifying and creating additional content to the game, user community and players can create custom mods and other expansions which can provide additional gameplay value for other players of the game [4 pp. 99-107]. In this section, some common approaches for providing AI customizability are considered from this project's point of view.

Black Box and White Box Approaches

The AI system implementation philosophies can be generally characterized into two groups: White Box and Black Box systems. The White Box systems offer more flexibility, are good for team of multiple people collaborating and they allow designers to work more independently. Black box systems on other hand are good for single-person implementation, but force designers to have greater dependency on programmers for providing implementations for the black-boxed behaviors. Neural Networks are one example of Black Box systems, which take discrete input and provide output through trained processing in the hidden layer [4 pp. 100-107].

Sometimes mixing the two is possible, for example reusable components in White Box systems requiring less customization are usually better suited as black box components, such as Pathfinding logic. The choice of better approach often depends on the project needs, for example implementing AI components as Black Box systems might be good for a team with many programmers, but in a team with many designers a White Box System would be more preferable [4 pp. 100-107].

3.8.1 Data-Driven Design

The key idea in Data-Driven Design is detaching the AI behavior and logic from the game code into a separate data model, which can be independently modified by the designer without need for programmer intervention. There are various ways of accomplishing this, for example with FSM state masks, custom parameter configurations for individual AI agents, external definition of rules for Rule-Based Systems and Scripting Languages [4 pp. 109-111].

Custom Tools

One way to increase the designers' power in AI development is through the development of custom tools for creation, debugging and visualization of AI in the Data-Driven Design model. The benefits of this are the increased flexibility, easier maintenance and balancing, and increased usability through UIs tailored specifically for the designers' needs. Drawbacks however include the upkeep required from the tool programmers and extra care needed for version control handling [36].

These tools can either be completely specific to the data required for a single use-case, or they can be more general, such as tools for visualizing and editing Decision Trees, Behavior Trees, or rules in Rule-Based Systems. The tools can either be external applications, or they can be built into the game, allowing adjusting of the AI data in real-time without need of exporting data and restarting the game [4 pp. 104-111].

3.8.2 Scripting Languages

One powerful way to provide access to implementation of AI is through a scripting system. They come with many benefits including [37]:

1. The programmers and designers can work in parallel, when designers are able to independently add behavior to game AI with scripting.
2. The simple syntax provided by scripting should be easy to use by the usually non-technically oriented designers.
3. Scripting is based on Data-Driven Design, in which the AI logic is separated from actual game code.
4. Increased AI development iteration speed through simpler AI implementation and reduced number of possible errors.
5. Safety offered by scripts running sandboxed or interpreted, reducing the possibility of causing game crashing through buggy scripts or maliciously crafted content.
6. Scripting offers great way to add extensibility and modding support to the game.

The drawbacks of scripting however include the need for script development tools for designers, and the possible performance overhead due to script interpretation at runtime. Both of these problems can be alleviated by using existing scripting tools which have already good tools for scripting, and are already optimized to be embedded as part of game [37].

Choosing a Scripting Language

Although there is no limit in which language can be used for scripting, there are a few most commonly used options in game development. Some game engines come with a built-in scripting system, such as UnrealScript in the Unreal Engine, and Torgue Script in the Torque Engine. The use of open-source scripting engines, such as Lua or Python,

is also very popular among game developers. Visual Scripting, in which the logic is visualized graphically to the designer instead of a text-based presentation, is also one possibility, and commercial libraries such as PlayMaker Visual Scripting for Unity3D exist to support this approach. And if none of these options is suitable for a particular game, developers can opt to create a completely custom scripting language and engine tailored for the particular needs of their use case [4 pp. 112-130; 38].

For this project, the choice of customization was narrowed to a combination of case-specific custom data models and behavior scripting using an open-source scripting engine. A custom script engine was out of the scope of this thesis, so the choice of scripting language was further narrowed down to selection between Lua and Python, with motivations explained in more detail below.

The Open Source Options: Lua and Python

Both Lua and Python are scripting engines with a good reputation of being well suited for embedding into games due to their free license, easy integration, and stable language specifications which are backed by strong existing developer communities [4 pp. 114-130]. As this project uses Unity3D and C# for implementation of the prototype, there are two popular frameworks which provide support for adding scripting languages on top of it; IronPython for Python scripting [39], and Moonsharp for Lua scripting [40].

In Lua, the language itself has been designed to be flexible and extensible, so although by default there is no support for Object-Oriented (OO) paradigm such as classes, inheritance and encapsulation, they can be added through the use of meta-tables. The interpreter itself does not support multithreading, so care has to be taken by either running the interpreter only in a single thread, using mutual exclusion for access control, or having multiple interpreters on separate threads. The language syntax itself is very similar to C-like languages with shallow learning curve, and the standard library is very small and easy to learn [17 pp. 449-450].

Python has native support for OO programming and it excels when it comes to mixing the script language with native languages. The language syntax depends heavily on indentation, but is generally considered one of the easiest languages to read and learn. There is a very large number of libraries available for Python, but in runtime the language suffers from size and speed issues. [17 pp. 451-452]

Based on the above considerations, Lua integration using MoonSharp was chosen for this project, main reasons being the easy integration and lightweight runtime which were key for the nature and scope of the prototype.

3.9 Cheating

As the majority of how the AI works is hidden from the player, there is often temptation to cut corners short either to artificially increase the level of difficulty provided by the AI, or perhaps just to save time in development. Some games are notorious for having cheating AI, and it may be a big spoiler for the gameplay experience and source of frustration, if the player feels that his opponent is exploiting an advantage he/she cannot match [6 pp. 3-4].

Some examples of cheating might be the ability of AI to ignore visibility status of map, thus having always full knowledge of the location and arrangement of all enemy units - or adding a certain resource production multiplier to the computer player's production, which would be tied to the AI difficulty level chosen by the player [41].

For this project, the use of features giving the AI player unfair advantage over human player is prohibited, and instead the difficulty variation is done through combination of regulating AI aggression on easier levels, adding randomness to the decision scoring, and otherwise tweaking the way AI makes choices based on the same information that is available to the player.

3.10 Performance Considerations

Unlike many other real-time game genres with strict performance requirements, the 4X strategy games have the benefit of being more relaxed when giving the AI processing time thanks to their turn-based gameplay model. However, care needs to be taken to balance between how much computing time is given to the AI to not cause too big slowdown in the gameplay, especially in late-game situations in larger game worlds when many AI agents owned by multiple AI players might be operating. This section introduces certain techniques to optimize and balance the processing time given to the AI features.

3.10.1 Execution Management

A key feature in managing the time consumed by AI is division of the AI logic into manageable tasks, and using some method to control the execution of them. In a single-threaded execution model there is a fixed maximum amount of time that can be spent for performing other tasks, as rendering of the game usually happens on the same thread. A general-purpose execution management system can be used to not only run the AI code, but to also control the time given to many other background tasks in the game such as asset downloading, audio and physics processing [17 pp. 693-725].

Scheduling

A basic scheduling approach is to assign tasks to be executed on certain frames, using a simple algorithm such as execution frequency. Figure 39 below shows how tasks A, B and C might be executed when scheduled on relatively prime frequencies.

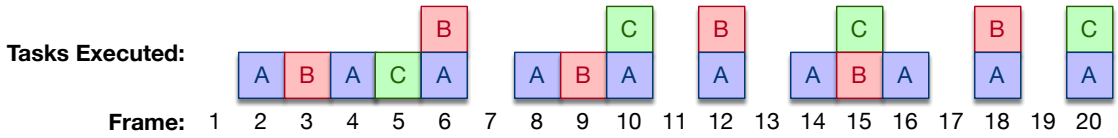


Figure 39. Frequency-based scheduling of tasks [17 p. 696].

The downside of this simplistic approach is the difficulty of estimating good frequency to prevent spikes caused by task clumping, although there are ways to alleviate that such as Wright's Method and Analytic Method. The frequency-based scheduling can be improved further by Priority-Based Scheduling, Load Balancing, or other means [17 pp. 693-725].

Hierarchical Scheduling

In Hierarchical Scheduling, the scheduler is able to not only run individual tasks, but also complete child scheduling systems inside itself. A possible organization of this type of scheduler is shown below in Figure 40.

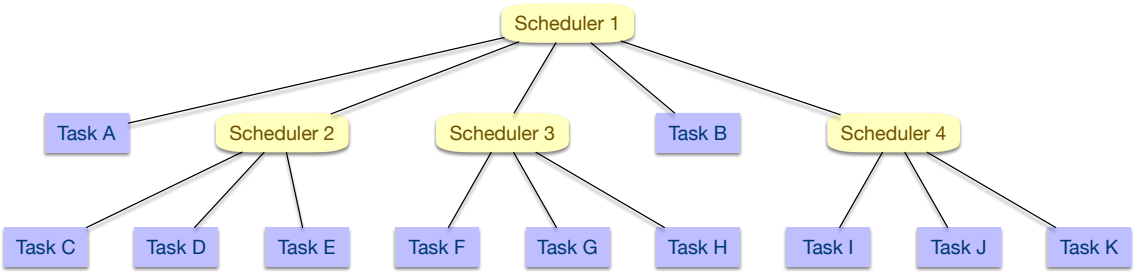


Figure 40. Example of a Hierarchical Scheduling System [17 p. 707].

The benefit of this approach is that the scheduling of certain behaviors can be isolated, which allows not only choosing the appropriate scheduling method for the child scheduler but also more modular design, in which parts of the AI can be replaced without causing major impact to the overall scheduling performance [17 pp. 693-725].

Load-Balancing

In a load-balancing system, the scheduler keeps track of time spent for each task, ensuring that processing time on each tick does not exceed the maximum limit. In addition, the expected running time of tasks can be estimated, which can be used to predict how long the task takes and thus help scheduling it for execution. Scheduling Groups can also be used to divide tasks into groups which share a certain scheduling algorithm, such as:

1. Spread Groups, in which tasks in a specified set are given an equal share of specified time to run after each other sequentially.
2. Count Groups, in which a specified number tasks are run per frame, regardless of their execution time.
3. Maximum Time Groups, in which the estimated time of tasks is used to schedule a set of tasks for the maximum duration specified.

Another benefit of load-balancing scheduler is that the profiling data of execution time is automatically available for debugging purposes [42].

Time-sliced pathfinding

One way to control the time consumed by pathfinding, and to prevent CPU spikes caused by excessively large pathfinding queries, is to divide a single path query operation to

multiple steps, a technique known as time-sliced pathfinding. With this approach, the pathfinder is given fixed amount of time to process during each tick, and when the time is exhausted the state of pathfinder is saved in such way that the query can be resumed when the pathfinder is again given processing time. One challenge with this approach is that if state of map affecting the result changes during the query, the results may not be guaranteed to be valid. How much this shortcoming affects the feasibility of the approach is however dependent on the use case of the pathfinding [43].

Multithreading

In the simple manual scheduling scenario, all tasks are run on a single thread in cooperative fashion. This requires both the AI tasks to behave well in terms of how long they take to execute on a single frame, and the scheduler has responsibility on how to divide the workflow during each tick. Another approach for allocating CPU time not only for AI tasks, but potentially to other components in the game engine too, is the use of multithreading. Traditionally, this approach has had the downsides of requiring synchronization of data structures and performance penalty caused by context switching, but in the past decade the advent of multicore CPUs even in the lowest-end mobile devices has created a situation, where the use of multithreading to leverage the new hardware features gives performance gains that outweigh the disadvantages [17 pp. 693-725].

3.10.2 GPU Offloading

Although the performance of Central Processing Units (CPUs) has increased constantly during the history of computers, the power of graphics processing units (GPUs) has exploded in the past decades to be a very viable option for performing large-scale computation. With the introduction of General-Purpose computation on the GPU (GPGPU) APIs such as OpenCL and DirectCompute, this power has become easily accessible to developers, including AI programmers. This allows great chance to speed up the AI by offloading applicable parts of it to the GPGPU processing [44].

As pathfinding is one of the core methods used in grid-based strategy games, seeking to optimize it through GPU offloading is one of the most promising applications for this technology. Recent research on has demonstrated that A*-type pathfinding can be implemented with GPGPU with a much higher performance compared to traditional CPU-based search [45]. The GPU offloading in Unity projects can be achieved with the use of

Compute shaders, which provides an API for running GPGPU programs within Unity applications. The downside of Compute shaders is they require using the latest versions of graphics programming APIs and graphics hardware supporting them [46].

3.11 Other Evaluated Methods

There are also a couple of methods which could be useful in the development of AI for strategy games, but they were considered not to fit in the scope of this project due to a combination of practical and schedule challenges. They do, however, have potential to be explored in future research, and are listed here to explain why they were rejected at this point.

Bayesian Networks

The Bayesian networks and Bayesian inference are focused on how the AI can deal with uncertainties, and make decisions based on probabilities of the unknown facts based on other known facts in the game state. These methods include [6 pp. 244-268]:

1. Diagnostic reasoning, which attempts to estimate the presence of a single source fact based on other existing facts in the world state which are known to be results of this source state.
2. Predictive reasoning, which attempts to estimate the presence of possible result fact from the known existence of a source fact.
3. Explaining away, in which a known result fact is used to reason the probability of presence of source facts leading to it, including characteristics of independence and conditional dependence.

The challenge in Bayesian networks is that for them to be useful, the probabilities used for inferring states need to be acquired by either gathering the information during simulated gameplay or through training of the network. This means that the problems they are used to solve need to be fairly uniform, as alteration of gameplay and rules affecting the probabilities causes the network to produce invalid decisions. One solution for this could be allowing the network to adapt to the user's gameplay by training it during actual gameplay, but this might make the AI show too much emergent behavior, and thus lead

to undesirable decisions which might not be expected by the AI programmer or programmer, causing other parts of the AI to behave unexpectedly and appear broken to the player.

Neural Networks

The concept of Neural Networks is one of the machine learning techniques which has gained a lot of popularity in the AI research in the recent years, and there are even some commercial games which have utilized this approach. The Neural Networks try to simulate the function of neurons in actual human brain with a simulated mathematical model. In this model, the Neural Network is divided into three layers: Input layer, Hidden layer and the Output layer, with each of those layers containing sets of neurons. The number on each layer dependent on the use case of the network. The source data is fed into the input layer, from which a feed-forward process passes the information through the hidden layer, all the way until it reaches the output layer. During training, the output data is evaluated and back-propagated to the weights in the hidden layer, and repeated until the Neural Network outputs the expected values. After this the trained network can either be used in the final AI implementation, or the training process can be infinitely continued to allow the AI to be able to adapt to player behavior during the gameplay [6 pp. 269-315].

The major challenge in the Neural Networks is, like previously outlined in the evaluation of Bayesian Networks, the need for training and high risk of unexpected emergent behavior, and these features make especially debugging, testing and balancing gameplay difficult [6 p. 271]. It should be noted that it is possible to leverage these techniques in games successfully, but due to the time and effort required for these approaches compared to the potential gain, they were not included in the scope of this thesis.

4 Proposed Solution

In this section, the produced high level technical design is presented in addition to outlining the best practices for implementing it. Due to the highly iterative process in game development, this solution should be considered as a starting point for prototyping, and most likely goes through various alterations and fine tuning as the development proceeds. It should however provide an understanding of what kind of options are being considered as realistic goals for the project.

4.1 Architecture Overview

On high level, the entire AI is encapsulated as the virtual player, which interfaces with the game by taking the world state as input, and providing set of actions as output. Figure 41 below shows an abstract overview of this information flow.

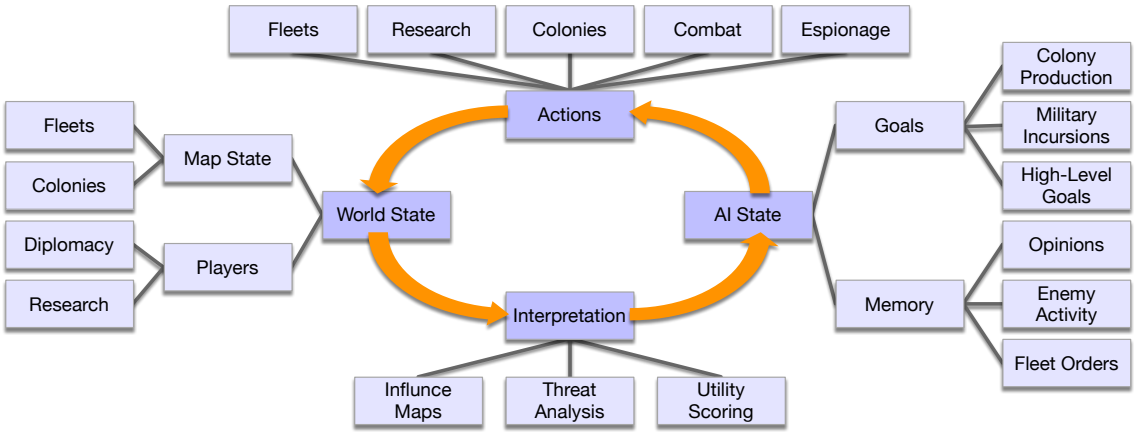


Figure 41. Abstract illustration of AI interaction with the game engine.

To give a better understanding of this process, below is a brief explanation of the purpose of each step in this illustration:

- The **World State** contains all information available to the AI player the explored map of the game world which contains all known colonies and fleets with their latest updated states, diplomatic relationships, research and other generally any other data available to each of the players during the gameplay.

- **Interpretation** step takes any data from the world state as input, and analyzes it to provide value for the AI player. This includes tactical analysis including influence mapping and threat analysis, and algorithms providing utility factors derived from the game state for various decision-making processes.
- The **AI State** encompasses any active goals and plans that AI has and any data required for them. Any other persistent information is also managed here, including memory of spotted units and opinions about other players.
- The AI interacts with the game by providing **Actions**, which include movement commands for fleets, choices made for colony production, and picking research goals and priorities. They are either initiated directly by the AI during the AI player's movement turn, or in response to certain game events occurring, such as battles and diplomatic meetings.

This abstraction of the AI's understanding of the game state and its operations allows better modularity, and helps preventing cheating by allowing the AI to use only the same information and actions which a human player would be able to utilize in the game.

Furthermore, with this architecture it is possible to use certain parts of AI features for human players to allow worker automation, colony governors and other features that might be automated to reduce the dreaded micromanagement which might be present in large-scale games.

4.2 AI Model and Components

When implementing the AI, a good approach for controlling the complexity is the division of the AI to dedicated components. The central part of the virtual player is the multi-tier AI model, which follows the principles set earlier in Chapter 3.2.1. The responsibilities of each tier level are capsulated to managers, each of which has a dedicated role in the decision-making hierarchy. These managers (and individual agents in lower tiers of the model) utilize various AI tools which are available for them throughout the tier levels as a separate toolbox. This division to components is illustrated below in Figure 42.

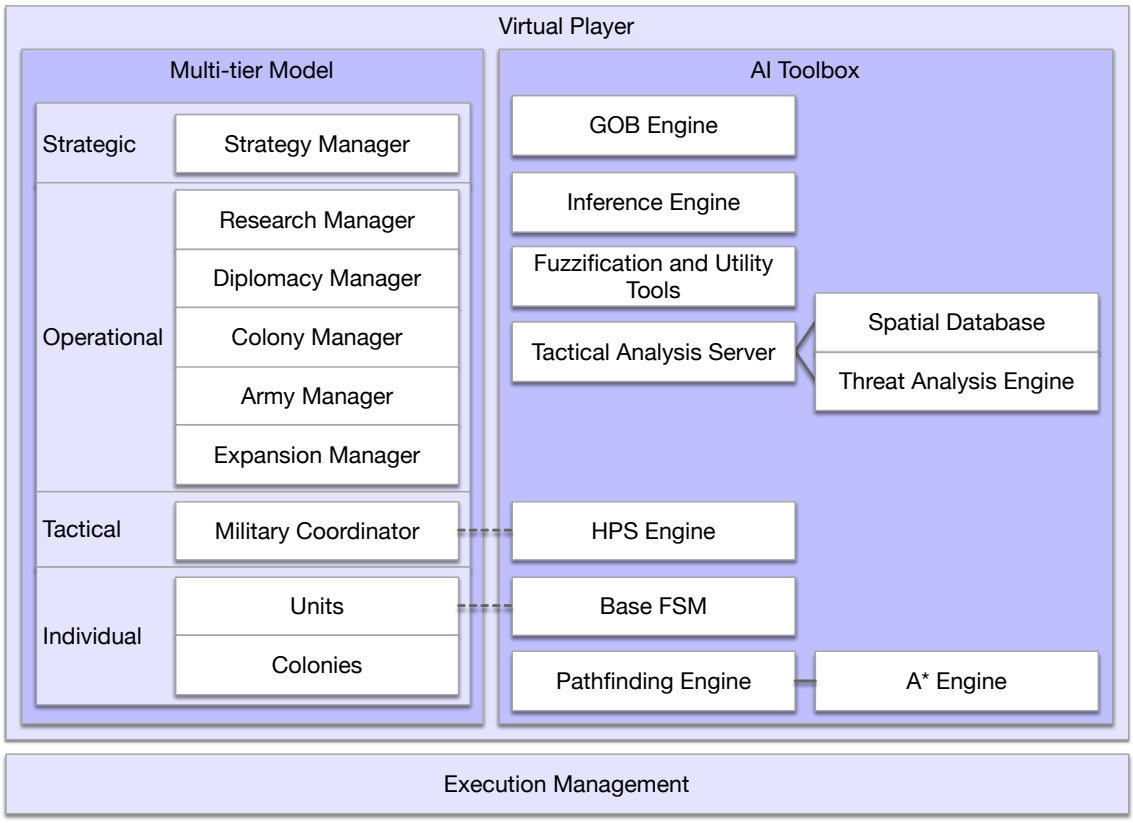


Figure 42. The core AI model and its components.

The roles of each component and their relationships are described briefly in the following chapters.

4.2.1 Strategic Tier

The highest level of the multi-tier model, and source of the decision making, is the strategic tier. All lower-level goals originate from this layer and affect all decision-making processes down in the hierarchy.

Strategy Manager

The root of AI decision making is built into the Strategy Manager, which uses a HTN planner to perform the high-level planning of the AI goals. The Figure 43 below shows the relationship between the Strategy Manager and other managers which it has authority over.

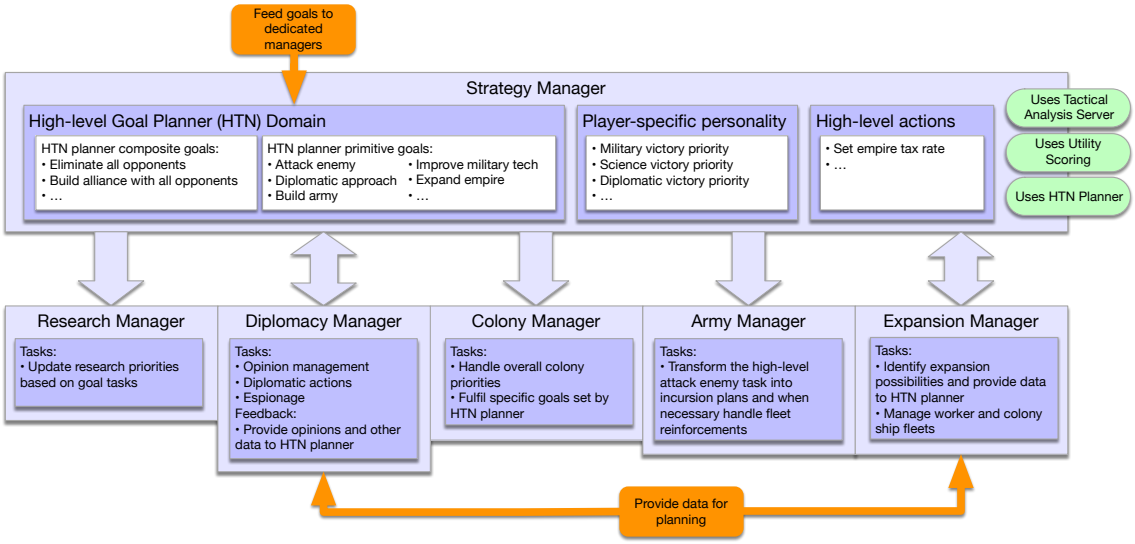


Figure 43. Overview of the Strategy Manager.

The HTN planner in this component utilizes a highly abstract, high-level strategic decision-making domain, which composes to various abstract tasks which are assigned to the other managers. The Diplomacy and Expansion Managers also provide data for the planner to help in the decision making.

4.2.2 Operational Tier

The next level of the multi-tier model is the Operational Tier, which contains manager components which are direct subordinates of the Strategy Manager. Each of them has the responsibility of both managing a dedicated subset of the AI behavior, and providing information necessary for the necessary higher-level decision manager.

Research Manager

The purpose of the Research Manager is to both choose which technologies the AI player should research, and to maintain knowledge of the enemy research state. The overview of Research Manager is shown below in Figure 44.

Figure 44. Overview of the Research Manager

The research goal selection uses in straightforward way the active goal given by the Strategy Manager to choose which field of research should be given the research priority. For tracking the state of enemy research, the Research Manager uses functionality of Rule-Based System's inference engine, allowing it to use knowledge of enemy fleet types, colony improvements, and existing knowledge of research state to further deduce what technologies that particular player is currently in possession of.

Diplomacy Manager

The Diplomacy Manager is responsible for handling any diplomatic actions requested by the Strategy Manager, and it also provides necessary data for the high-level HTN planner, such as the opponent reputation based on opinion values. The overview of the Diplomacy Manager is shown below in Figure 45.

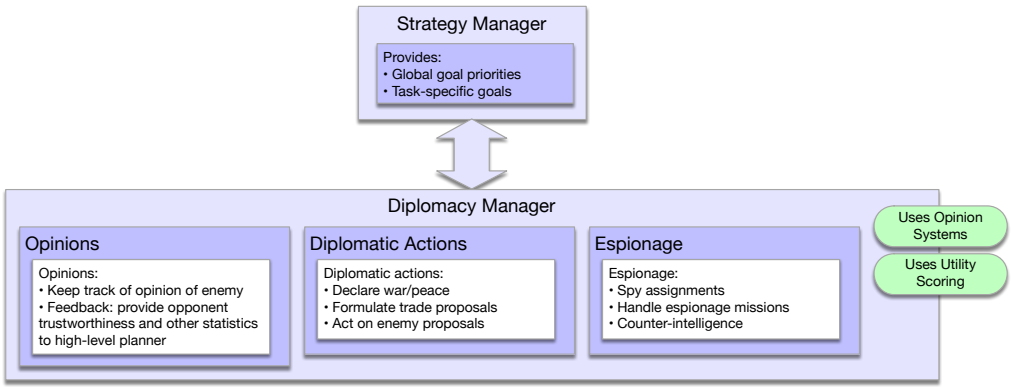


Figure 45. Overview of the Diplomacy Manager.

This manager itself has a threefold function:

- **Managing opinions**: Use of an opinion system to keep track of player's opinion about other players.
- **Perform diplomatic actions**: Includes declaring war and offering peace treaties, using utility scoring to compose any trade proposals, acting on enemy proposals, etc.
- **Handle espionage**: This includes assigning spies based on utility scoring, initiating espionage missions, and handling counter-intelligence resourcing in colonies

Colony Manager

The Colony Manager is responsible of controlling the production and other aspects of all colonies currently owned by the AI player. The overview of the Colony Manager is shown below in Figure 46.

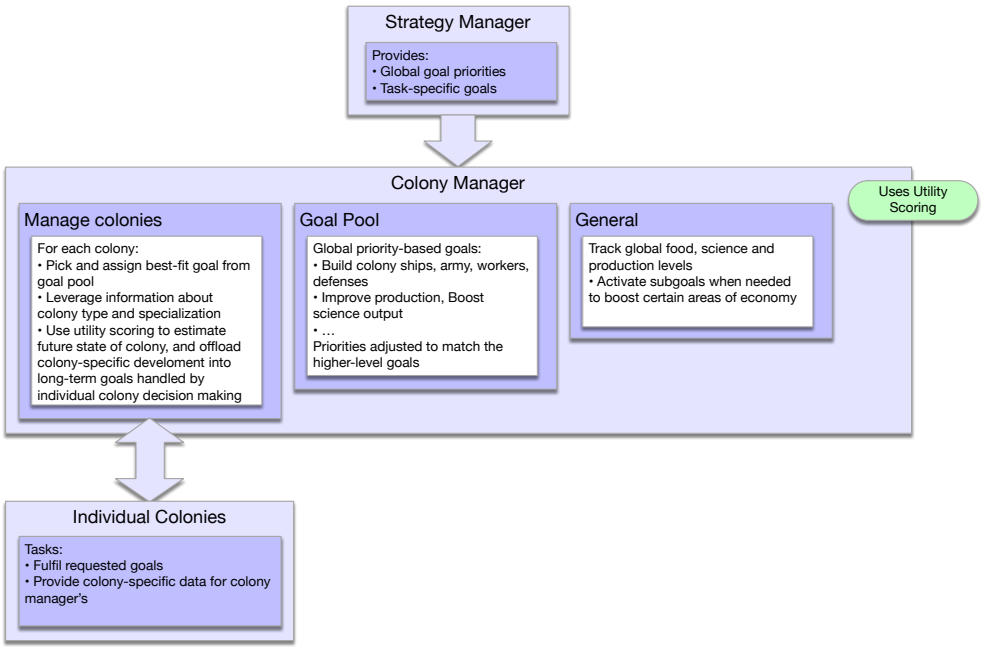


Figure 46. Overview of the Colony Manager.

One central component of the Colony Manager is the goal pool; it contains all possible goals, and at any given time each colony is assigned one of these goals which it needs to fulfill. As the Colony Manager gets the high-level goals from the Strategy Manager, it makes sure that the currently assigned goals have the highest utility for reaching this high-level goal. As the Colony Manager is also responsible of tracking global resource

production and allocation, this allows it to not only directly respond to specific production goals, but also to balance the resources between individual colonies to avoid potential production or growth bottlenecks.

Army Manager

The Army Manager is responsible for both high-level planning of military missions, and providing military data for the Strategy Manager. The overview of the Army Manager is shown below in Figure 47.

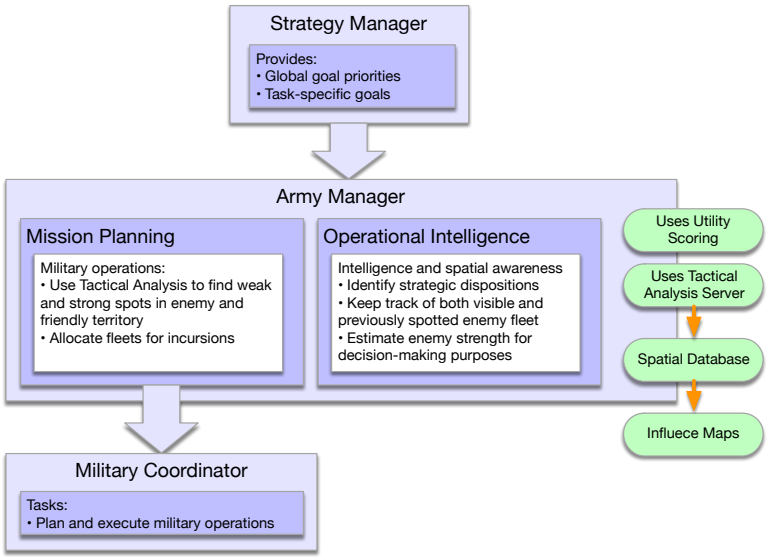


Figure 47. Overview of the Army Manager.

The mission planning relies strongly on the Tactical Analysis Server to find both weak and strong areas in the friendly or enemy territories, which allows it to create missions not only for incursions to enemy space, but also for reinforcing own defenses. This planning process also includes allocation of available military units to specific missions. When a potential mission is formulated, the allocated units and the mission goal is forwarded to the military coordinator, which handles planning of the actual mission and decomposition to individual unit actions.

Expansion Manager

The Expansion Manager is responsible for handling both the expansion of player's empire with colony ships, and using worker fleets to improve the existing structures owned by it. The overview of the Expansion Manager is shown below in Figure 48.

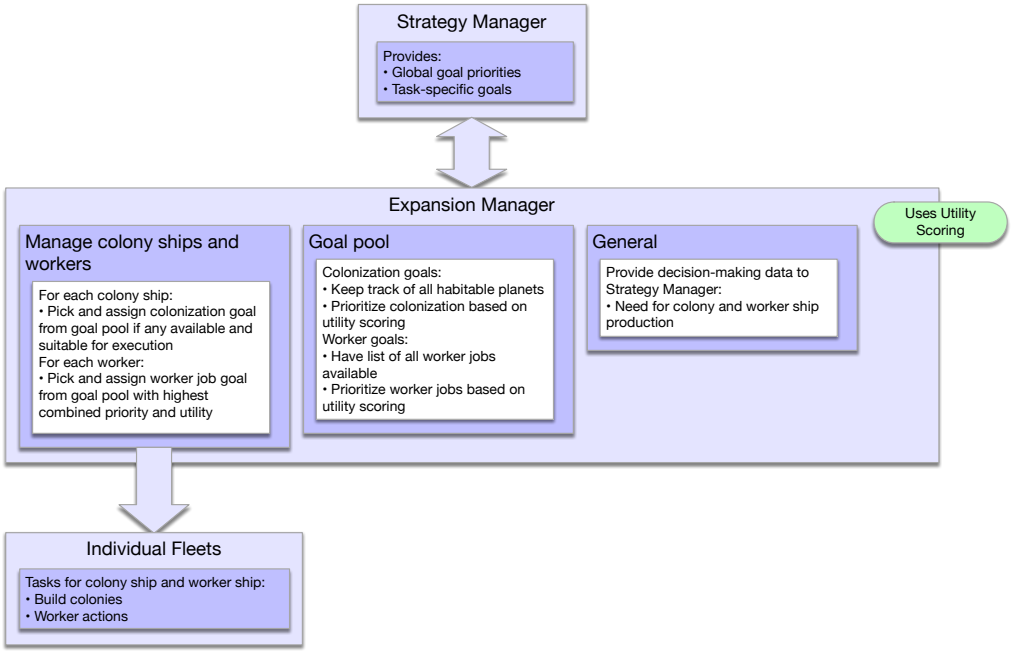


Figure 48. Overview of the Expansion Manager.

There are two responsibilities which the Expansion Manager handles:

- **Colonization**: List of all habitable planets is used in combination with evaluation of the utility of inhabiting of each world. This allows not only keeping a pool of colonization goals, from which the most potential ones can be assigned to existing colony ships, but also to identify need of new colony ships to be produced, which can be signaled to the higher-level Strategy Manager.
- **Workers**: Alike with colonization goals for colony ships, the complete list of potential structure construction and improvement actions is kept in order to provide worker units job goals, and also to evaluate the need for new worker unit production.

The goals given to colony ships and worker units are directly assigned to the individual units, which use their own decision-making model to fulfill them.

4.2.3 Tactical Tier

The role of the Tactical Tier in the multi-tier model is to manage tactical planning, which in the case of this 4X strategy game AI model is solely for coordination and planning of combined military maneuvers.

Military Coordinator

The purpose of the Military Coordinator is to take the mission goal and allocated units from the Army Manager, and to translate them into tasks for each of the individual units. Structure of this manager is shown below in Figure 49.

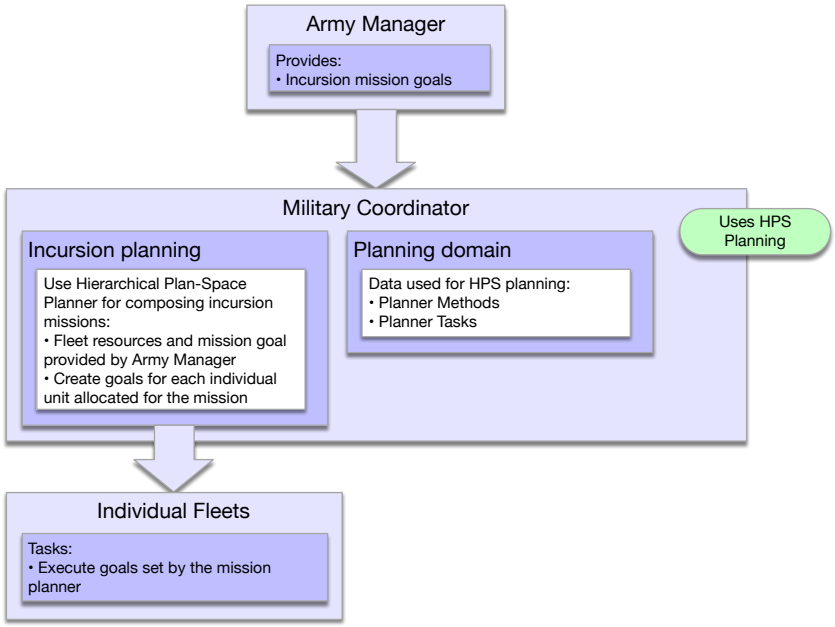


Figure 49. Overview of the Military Coordinator.

The core of Military Coordinator is the Hierarchical Plan-Space (HPS) Planner, which uses the planner methods and tasks provided by the coordinator's data model. Functionality of this HPS planning process was detailed earlier in Chapter 3.6.4. If the planning succeeds, the resulting plan containing tasks for each individual fleet is executed by the coordinator, which provides goals for the individual units during the plan execution until mission is either finished or failed.

4.2.4 Individual Agents Tier

The lowest level on the multi-tier model contains the behavior of individual AI agents, which in this game are units and colonies which the player owns.

Individual Fleets

Each unit in the game has an active goal, which is assigned to them either by the Military Coordinator, or the Expansion Manager, depending on the type of fleet in question. Overview of the Individual Fleets module is shown below in Figure 50.

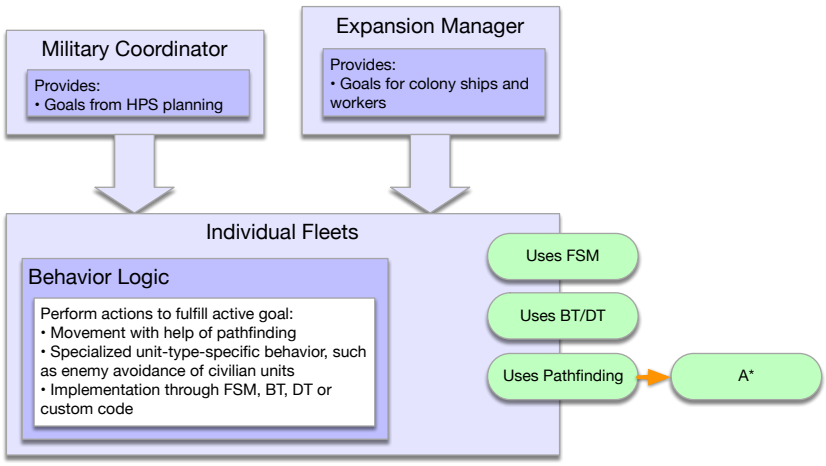


Figure 50. Overview of the Individual Fleets.

The fleet behaviors can utilize for example Finite State Machines (FSMs), Behavior Trees (BTs) or Decision Trees (DT), a combination of them, or custom code. There can be different behavior profiles depending on the unit type, allowing for example civilian units (workers, colony ships) and other weaker units to prefer avoiding the enemy. This module also leverages the Pathfinding component, which allows the units to perform movement on the game map.

The decisions made at this level are mapped directly to the commands given to fleets during the AI player's unit movement turn during the gameplay. The worker automation can be also used by the human player to toggle partial Expansion Manager and AI fleet behavior on and off when needed.

Individual Colonies

Each colony owned by the AI player is managed by the Individual Colonies AI module, which takes the goals given by the Colony Manager, and translates them into actions that can be performed by the AI for that particular colony. Overview of the Individual Colonies module is shown below in Figure 51.

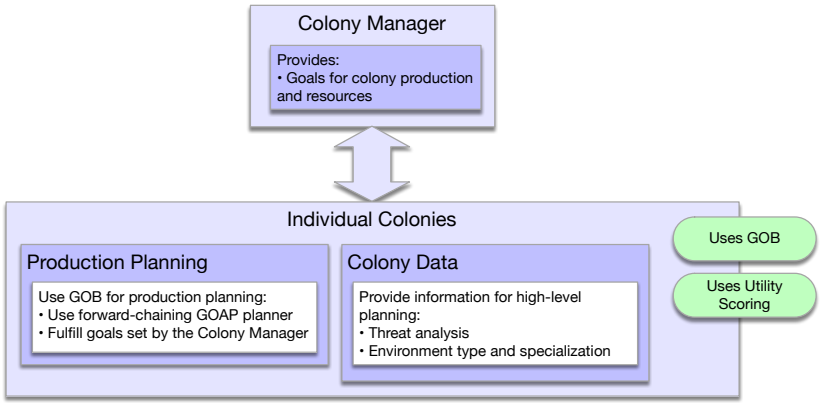


Figure 51. Overview of the Individual Colonies.

There are two main ways the AI can control the colony:

- Setting active production
- Changing colony resource allocations

Both of these can be changed through goals available for the colony decision-making, which utilizes a Goal-Oriented Behavior (GOB) implemented as a forward-chaining Goal-Oriented Action Planner (GOAP). Contrary to regular GOAP, which traverses back from the goal state to initial state, the colony production planning should use the forward-chaining method because certain actions, such as building factories, affect the scoring of actions performed later in chronological order.

Another role that the Individual Colonies model has is evaluating the threat level of each particular colony, and also any colony-specific specializations and other data needed for the higher-level Colony Manager decision-making process.

4.2.5 AI Toolbox

The AI toolbox contains various generic AI algorithms and tools, introduced earlier in the chapter 3, which are utilized by the various other modules throughout the multi-tier levels of the AI decision-making model. The most important principle of the toolbox is that each of the technologies used by the AI is encapsulated into a reusable component, which allows integration with other AI code through predefined interfaces. This not only allows keeping the component-specific code separate from other AI logic, but also allows better possibility for scripting language integration, explained later in Chapter 4.4.

4.2.6 Execution Management

Unlike in real-time strategy games, which have strict limits on the CPU budget for AI execution time per frame, the nature of turn-based games allows certain level of flexibility in the implementation of AI processing model. Although not optimal, in the proposed prototype, all AI actions can be isolated to the AI players' turn, thus reducing the pressure to interleaving AI processing with other players' turns. However, even when executing all AI code of a particular player in one batch, certain steps must be taken to avoid undesirable behavior of the game, such as stalling the game interface during the AI processing.

In Unity, the default method of implementing asynchronous behavior is through the use of "Coroutines" which resemble cooperative multitasking. Each Coroutine gets executed on the Unity's main thread, and they are responsible for handing over the execution voluntarily by yielding. Although this approach is easy to implement, it has the drawback of needing careful planning of when to yield the execution to avoid either clogging too much CPU time on a single Coroutine pass, or wasting CPU time by yielding excessively often. Other challenge of this approach is the care needed when designing the AI model to allow Coroutine yielding throughout different components.

Another way of implementing the asynchronous execution is through the use of C# threads. Although the use of threading allows independent execution from the main thread, and possibility of benefiting from multi-core CPUs, they have the drawback of needing synchronization between any shared data structures, and as Unity's API is not thread-safe, would be limited to executing only the parts of C# code which are not using Unity features.

In this prototype, the Coroutine method was chosen to be used.

4.3 AI Diagnostic Tools

The Unity Editor can be extended easily by custom tools, a feature which can be leveraged to create tools that allow rich and verbose view to the state and working of the AI while it is executing in the game running in the play mode.

4.3.1 Automated Testing

When balancing the AI behavior and features, it would be unnecessarily time consuming to have QC play multiple games against an AI after each modification done to it by the programmers or designers. To alleviate this problem, automated testing can speed up a lot of this process by allowing simulation of entire games played by variable number of AI players. A set of parameters can be configured for this process:

- Skill level of the AI player as the human player would select it
- “Personality” parameters such as AI temperament, diplomatic behavior, and overall goals
- Any basic game settings such as the galaxy size, number of players, victory conditions

Also, different variations of AI using their own versions of data models could be potentially matched against each other, to directly evaluate the effect of modifications of the model data on AI performance with minimal iteration time.

Each automated game produces a transcript of all actions performed during the game, and the resulting outcome of the game. This recorded journal of the game can be replayed in either high- or low-level detail, which allows designers to pinpoint which actions by the AI were undesired, and give them ideas which features would need to be improved.

4.4 Scripting Using LUA

In the proposed AI model, the individual manager modules in the multi-tier have the potential to benefit from being abstracted from the other AI code through use of LUA scripting engine. The AI toolbox, which have very little need (if any) for specialization, can be easily utilized by scripts as standalone components exposed to the scripting environment, removing the need to write any low-level AI code in the scripting language.

As explained in Chapter 3.8.2, the MoonSharp library was chosen to be used in the prototype, but due to schedule challenges the experimentation of scripting integration was not finalized in time for this thesis, and remains on theoretical level.

4.5 Data Model for AI

Following the principles of the Data Driven Design introduced in Chapter 3.8.1, many parts of the AI model are isolated into data models, which can be customized either by custom tools or direct manipulation of the data in question. These datamodels are listed below in Table 13.

Table 13. Table of data models.

Model type	**Individual model use cases**
HTN Domain	• Strategy Manager HTN planner
Opinion System Model	• Diplomatic decision making
Rule-Based System Rules	• Research Manager technology inference
Utility Model	• High-level Strategy Manager decisions • Part of Strategy Manager HTN planning • Diplomatic decision-making • Espionage decision-making • Colony Manager global resource allocation • Colony Manager goal prioritization • Colony resource allocation • Army Manager operational intelligence • Expansion Manager production desirability
Spatial Database	• Rules for combining influence data into spatial database by the Tactical Analysis Server
HPS Domain	• Military Coordinator Hierarchical Plan-Space planner
Decision Tree	• Potential use for individual fleet decision-making
Behavior Tree	• Potential use for individual fleet decision-making
Goal pool	• Colony Manager colony goals • Expansion Manager fleet goals
GOAP Goal Hierarchy	• Colony production GOAP planning

All of the above data models can be stored in either JSON or XML presentation, which can be further exposed to the designers with an UI through custom tools where needed. This allows building a foundation for the AI behavior before devoting time to building the custom tools. It should be noted that some data, such as the rules used for technology inference, can be derived from other game data.

5 Evaluation

This chapter explains what was developed during the project, and evaluates how well the project output matched the initial goals. The biggest challenge for the implementation of the complete AI player was the lack of a 4X game engine, and creating one single-handedly for the purpose of this thesis would have exceeded the feasible time limits available to finish the project, and risked generating excessive workload beyond the scope of the project. The outcome of this thesis can be categorized in two distinct parts:

- High-level technical design
- Code implementation

Both areas of output contribute to the results of this thesis, as explained further in the next chapters.

5.1 High-level Technical Design

The most important outcome of the project was the high-level design of the AI player, which should be used as a starting point and guideline for the implementation of the actual AI code in the actual game as presented earlier in Chapter 4 as the proposed solution. This technical design contributed to the selection of parts suitable for independent prototyping, which are detailed later in Chapter 5.2.

It should be noted that the greatest value of the high-level design comes from this profound research of AI field done for the theory part of this thesis, as the time spent on this process will be saved in any possible future implementation of actual games.

As the evaluation of this technical design was limited to general assessment of its usefulness in the scope of this thesis due to lack of a complete game prototype, the results of the prototyping were used to reinforce and assess the feasibility of this high-level design.

Although the technical design represents the virtual player feature of the strategy game, all necessary parts of the design can be applied for player-assisting “governor” features by considering the human player as a complete AI player with certain decision-making modules disabled. This allows the full influence mapping, threat evaluation and other

features to provide necessary data for the assisted features, such as production planning, advisor recommendations, scientific research goal suggestion, etc.

5.2 Technology Prototyping

As the full-scale implementation was ruled out, the prototyping phase focused on elevating of certain key technologies which were part of the high-level design. All prototyping was done using Unity 5.6 game engine on macOS, which also was the target platform for the proposed solution. As part of the process, source code was produced which is available as an appendix in this thesis:

- Appendix 5 contains the latest version of the prototype of map testing application, which combines pathfinding and spatial database experiments in one program.
- Appendix 6 contains source code for the inference engine prototype.

These appendices only contain the C# source code, and do not include actual Unity scenes and asset files which were used in the prototyping.

5.2.1 Pathfinding with Generic A* Engine Prototype

As per the original game design, the map prototype was based on the hexagonal grid. This grid type provides better movement and distance handling by eliminating the different between diagonal and axial distances, but in contrast requires more complicated handling of hexagonal coordinates. A hex coordinate utility was developed during the project to help managing this challenge.

Appendix 1 shows screenshots of the first pathfinder prototype. In the pictures, there are the following features:

- Yellow hex indicates start node of pathfinder.
- Black is the goal node for pathfinder.
- Yellow dots represent the final path of the query.
- Hexes with black borders indicate which nodes were evaluated during each query
- The hex cell color indicates traversal cost of each node, light blue being the highest-cost path, and purple the lowest-cost.

- Additionally, the white openings in the map indicate blocked nodes which cannot be passed, as demonstrated in the bottom-left screenshot which shows result of a query in which a path could not be found.

This first version provided a good foundation for the map and pathfinder code, although some bugs were found and fixed in the later version.

5.2.2 Spatial Database and Influence Mapping Prototype

Building upon the map prototype used for the pathfinding testing previously, the functionality was extended to include support for multi-level spatial database to map player influence and the combined effects for visualization on the map. Additionally, the pathfinding engine was adapted to leverage the spatial database information to allow experimenting with tactical pathfinding.

Appendix 2 shows screenshots of the various spatial database layers being visualized. This version of prototype was equipped with a primitive representation of military units, which appear as circles on the map. Each of the units has the following properties:

- **Strength**: used in influence calculation to determine power and distance of the effect, shown as number on the unit.
- **Friendly flag**: used to distinguish player's own units from enemy units; green circles represent friendly units, red are enemy units.

The screenshots show the following layers in the spatial database:

- **Movement cost**: This layer contains the movement cost of each map cell. In the screenshot, green cells have the lowest cost, while red ones have the highest cost.
- **Own influence**: In this layer, all friendly units are used as influence sources, which propagate their influence based on their strength. The influence is spread linearly, with each increase in strength causing the influence to spread one hex further away from the source unit. This effect is additive, so nearby units enforce each other's influence. In the screenshot, green area shows strong own influence.

- **Enemy influence**: Working like the own influence layer, the enemy influence layer shows the propagated influence of units, but for enemies. In the screenshot, red area shows strong enemy influence.
- **Combined influence**: This layer combines the two previous layers, own influence and enemy influence, so that areas with own influence have positive values, and ones under enemy influence have negative values. In the screenshot, green areas have strong own influence, while red areas are under enemy influence.
- **Tension**: This layer combines the own influence and enemy influence by summing them instead of subtraction, thus creating higher values in areas with lot of military activity. In the screenshot, green areas have no tension, while red areas have high tension value.
- **Vulnerability**: The vulnerability layer is calculated by subtracting the absolute value of combined influence from the tension layer. This allows identifying units on either side, which are exposed to strong opponent force, and thus making themselves vulnerable. In the screenshot

The aforementioned spatial database layers were used just for testing purposes, and actual game could use any number and any combination of layers in whichever way the game designers might feel useful.

5.2.3 Tactical Pathfinding

Tactical pathfinding was created as a simple extension to the original pathfinder, using the influence data in the spatial database to show benefits of this approach. The screenshots in Appendix 3 show three different pathfinding modes with equal start and goal locations:

- **Simple**: Each node is considered to have fixed cost of 1, which makes the pathfinder attempt finding the route with lowest number of map cells from start to the goal node. The screenshot also shows, that this approach makes the pathfinder visit very low number of map cells thanks to the A* algorithm which prioritizes cells closer to the target.
- **Terrain movement cost**: The pathfinder now considers movement cost of terrain when doing the path query, showing a different resulting route. Also, a much higher number of map nodes was visited during this query.

- **Enemy threat avoidance**: Again, the same start and goal nodes are queried, but this time the scoring of path is done using data from the enemy influence layer (although boosted 10 times to allow including terrain movement cost as a secondary scoring method), which creates a completely different path, which nicely skirts around the areas around enemy influence.

The actual formulas for combining influence data from spatial database for map cells are highly customizable, and may need a lot fine-tuning depending on the use case for the game. For example, it may need careful consideration on how much effort the AI should put into avoiding enemy if resulting path leads to excessively long detour. However, for the prototype purpose, the combinations used seem to work quite well.

5.2.4 Inference Engine Prototype

A simple inference engine was created to demonstrate the basic concepts of rule-based system functionality. This implementation uses heavily Unity features, such as storing rulesets in ScriptableObjects, and editor extensibility to allow running the inference engine in either edit or runtime mode.

Screenshots of the inference engine test are shown in Appendix 4, which were taken from the Unity Editor mode. The first screenshot shows the ruleset asset, and the associated test rules used in the prototype. The other screenshots show state of the inference engine after each user action, starting from resetting the inference engine followed by single-stepping it one step at a time until the inference process was finished. The debug output of the Unity console is also included in the screenshots, which shows the effects of inference process during each step.

Although the basic inference process works, and demonstrates the functionality of associated algorithms, a more generic implementation supporting dynamic rule generation would probably be most useful for the actual production-quality game, especially if used for technology inference like proposed in the high-level design.

6 Discussion and Conclusions

The motivation for this thesis originated from the interest in game artificial intelligence and the 4X strategy gaming genre. This, combined with the ongoing trend of constantly increasing mainstream popularity of gaming, gave further incentive for this research as a way to create a design which could benefit future development projects requiring this type of knowledge.

As possibilities for exploring this field of technology were not available at the workplace for the timeframe of this thesis, the project was implemented as a personal undertaking focusing strongly on theoretical research.

The main goal of this thesis was to research various AI technologies in order to create a plan for integrating AI for a turn-based 4X strategy game, and a secondary goal was to do prototype implementation of it. The theoretical part of the study was finished, but the implementation was limited to prototyping only certain areas of the researched technologies.

The technology research phase proved highly informative, and a lot of knowledge was gained for creating a feasible high-level design for the AI implementation. During the prototyping phase, the technologies that were experimented with also showed a lot of their potential also in practical setting, thus reinforcing the credibility of the parts of design which they were associated with.

Although there were certain shortcomings in the practical output of the thesis, the theoretical knowledge gathered has high value for not only the author personally as game developer, but hopefully also for other people facing similar challenges who might be reading this.

One major improvement in future would be the full implementation of AI in the actual game. As the AI design is theoretical and high-level, there are most likely many challenges in the practical implementation which will be reflected back on the design of the AI model. This includes for example considering the actual gameplay design of the game and the processing performance of the target platform.

Also, the further work should not be limited by the research done in this thesis, but should also be open to exploring possibilities of both future technologies, and re-evaluating the existing technologies, such as Bayesian and Neural Networks which were ruled out of the scope for this thesis.

References

1. **Entertainment Software Association.** *ESA Annual Report 2015.* [Online] 2016. [Cited: March 31, 2017.] http://www.theesa.com/wp-content/uploads/2016/04/ESA-Annual-Report-2015-1.pdf.

2. **Flew, Terry and Humphreys, Sal.** Games: Technology, Industry, Culture. [book auth.] Terry Flew. *New Media: An Introduction.* 2nd Edition. South Melbourne : Oxford University Press, 2005, p. 106.

3. **Baekkelund, Christian.** Academic AI Research and Relations with the Game Industry. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Boston : Thomson Delmar Learning, 2006, Vol. 3, pp. 77-86.

4. **Ahlquist, John and Novak, Jeannie.** *Game Development Essentials: Game Artificial Intelligence.* s.l. : Thomson Delmar Learning, 2008. ISBN-10: 1-4180-3857-1.

5. **Tozour, Paul.** The Evolution of Game AI. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 3-14.

6. **Bourg, David M. and Seemann, Glenn.** *AI for Game Developers.* Sebastopol : O'Reilly Media, 2004. ISBN: 978-0-596-00555-9.

7. **Emrich, Alan.** MicroProse's Strategic Space Opera is Rated XXXX! *Computer Gaming World.* 1993, 110, pp. 92-93.

8. **Wikipedia contributors.** 4X. *Wikipedia.* [Online] February 23, 2017. [Cited: February 27, 2017.] https://en.wikipedia.org/wiki/4X.

9. —. Civilization (series). *Wikipedia.* [Online] February 4, 2017. [Cited: February 27, 2017.] https://en.wikipedia.org/wiki/Civilization_(series).

10. —. Master of Orion. *Wikipedia.* [Online] December 5, 2016. [Cited: February 27, 2017.] https://en.wikipedia.org/wiki/Master_of_Orion.

11. —. Galactic Civilizations. *Wikipedia.* [Online] February 28, 2017. [Cited: March 1, 2017.] https://en.wikipedia.org/wiki/Galactic_Civilizations.

12. **Stardock.** Galactic Civilizations Gold. *Stardock Corporation.* [Online] 1998. [Cited: March 10, 2017.] http://www.stardock.com/products/gcgold/.

13. **Wardell, Brad.** Stardock's OS/2 history. *Stardock Corporation.* [Online] [Cited: March 28, 2017.] http://www.stardock.com/stardock/articles/article_sdos2.html.

14. **Unity Technologies.** Games | Made With Unity. *Unity - Game Engine.* [Online] [Cited: April 7, 2017.] https://madewith.unity.com/en/games.

15. —. Editor Version Release Dates. *Internet Archive.* [Online] October 15, 2014. [Cited: April 7, 2017.] http://web.archive.org/web/20141015144227/http://docs.unity3d.com/Manual/ReleaseDates.html.

16. —. Unity - Editor. *Unity - Game Engine.* [Online] 2017. [Cited: March 21, 2017.] https://unity3d.com/unity/editor.

17. **Millington, Ian.** *Artificial Intelligence for Games.* San Francisco : Morgan Kaufmann, 2006. ISBN: 978-0-12-497782-2.

18. **Levy, David N. L.** *Computer Gamesmanship.* New York : Ishi Press International, 1980. ISBN: 4-87187-805-8.

19. **Ramsey, Michael.** Designing a Multi-Tiered AI Framework. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2004, Vol. 2, pp. 457-466.

20. **Matthews, James.** Basic A* Pathfinding Made Simple. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 105-113.

21. **Dawe, Michael, et al.** Behavior Selection Algorithms: An Overview. [ed.] Steve Rabin. *Game AI Pro.* Boca Raton : CRC Press, 2014, pp. 47-60.

22. **Apple Inc.** GKDecisionTree Class Reference. *Apple Developer Documentation.* [Online] 2017. [Cited: June 11, 2017.] https://developer.apple.com/documentation/gameplaykit/gkdecisiontree.

23. *How Behavior Trees Modularize Hybrid Control Systems and Generalize Sequential Behavior Compositions, the Subsumption Architecture, and Decision Trees.* **Colledanchise, Michele and Ögren, Petter.** 2, April 2017, IEEE Transactions on Robotics, Vol. 33, pp. 372-389.

24. **Merrill, Bill.** Building Utility Decisions into Your Existing Behavior Tree. [ed.] Steve Rabin. *Game AI Pro.* Boca Raton : CRC Press, 2014, pp. 127-136.

25. **Graham, David "Rez".** An Introduction to Utility Theory. [ed.] Steve Rabin. *AI Game Pro.* Boca Raton : CRC Press, 2014, pp. 113-126.

26. **Harmon, Vernon.** An Economic Approach to Goal-Directed Reasoning in an RTS. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 402-410.

27. **Sweetser, Penny.** Environmental Awareness in Game Agents. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Boston : Charles River Media, 2006, Vol. 3, pp. 457-648.

28. **Tozour, Paul.** Using a Spatial Database for Runtime Spatial Analysis. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2004, Vol. 2, pp. 381-403.

29. **Woodcock, Steven.** Recognizing Strategic Dispositions: Engaging the Enemy. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 221-232.

30. **Straatman, Remco, Beij, Arjen and van der Sterren, William.** Dynamic Tactical Position Evaluation. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Boston : Charles River Media, 2006, Vol. 3, pp. 389-403.

31. **Orkin, Jeff.** Applying Goal-Oriented Action Planning to Games. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2004, Vol. 2, pp. 217-227.

32. **Humphreys, Troy.** Exploring HTN Planners through Example. [ed.] Steve Rabin. *Game AI Pro.* Boca Raton : CRC Press, 2014, pp. 149-167.

33. **Dybsand, Eric.** Goal-Directed Behavior Using Composite Tasks. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2004, Vol. 2, pp. 237-245.

34. **van der Sterren, William.** Hierarchical Plan-Space Planning for Multi-unit Combat Maneuvers. [ed.] Steve Rabin. *Game AI Pro.* Boca Raton : CRC Press, 2014, pp. 169-183.

35. **Russell, Adam.** Opinion Systems. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Boston : Charles River Media, 2006, Vol. 3, pp. 531-554.

36. **Snavely, P. J.** Custom Tool Design for Game AI. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Boston : Charles River Media, 2006, Vol. 3, pp. 3-12.

37. **Tozour, Paul.** The Perils of AI Scripting. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 541-547.

38. **Hutong Games.** PlayMaker Visual Scripting for Unity3D. *Hutong Games.* [Online] 2017. [Cited: June 17, 2017.] http://hutonggames.com/index.html.

39. **IronPython Community.** IronPython. *IronPython.* [Online] 2017. [Cited: June 17, 2017.] http://ironpython.net/.

40. **Mastropaolo, Marco.** MoonSharp. *MoonSharp.* [Online] 2017. [Cited: June 17, 2017.] http://www.moonsharp.org/.

41. **Johnson, Soren.** The Unique Challenges of Turn-Based AI. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2004, Vol. 2, pp. 399-403.

42. **Bob, Alexander.** An Architecture Based on Load Balancing. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 298-304.

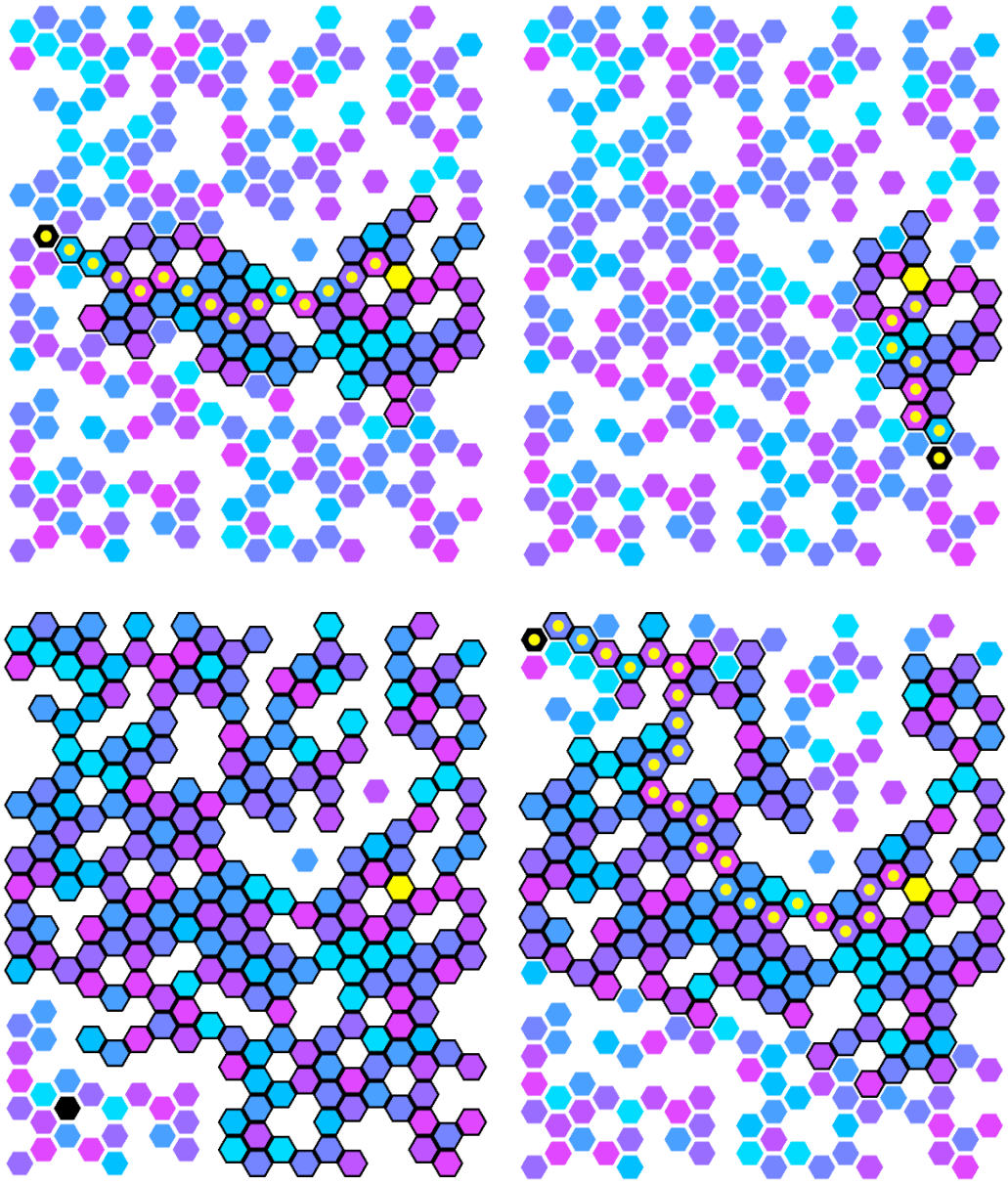
43. **Higgins, Dan.** Pathfinding Design Architecture. [ed.] Steve Rabin. *AI Game Programming Wisdom.* Hingham : Charles River Media, 2002, Vol. 1, pp. 122-132.

44. **Bourke, Conan and Bednarz, Tomasz.** Introduction to GPGPU for AI. [ed.] Steve Rabin. *AI Game Pro.* Boca Raton : CRC Press, 2014, pp. 539-547.

45. *GPU accelerated pathfinding.* **Bleiweiss, Avi and NVIDIA Corporation.** Sarajevo : Eurographics Association, 2008. GH '08 Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware. pp. 65-74. ISBN: 978-3-905674-09-5.

46. **Unity Technologies;.** Compute Shaders. *Unity Documentation.* [Online] June 2, 2017. [Cited: June 13, 2017.] https://docs.unity3d.com/Manual/ComputeShaders.html.

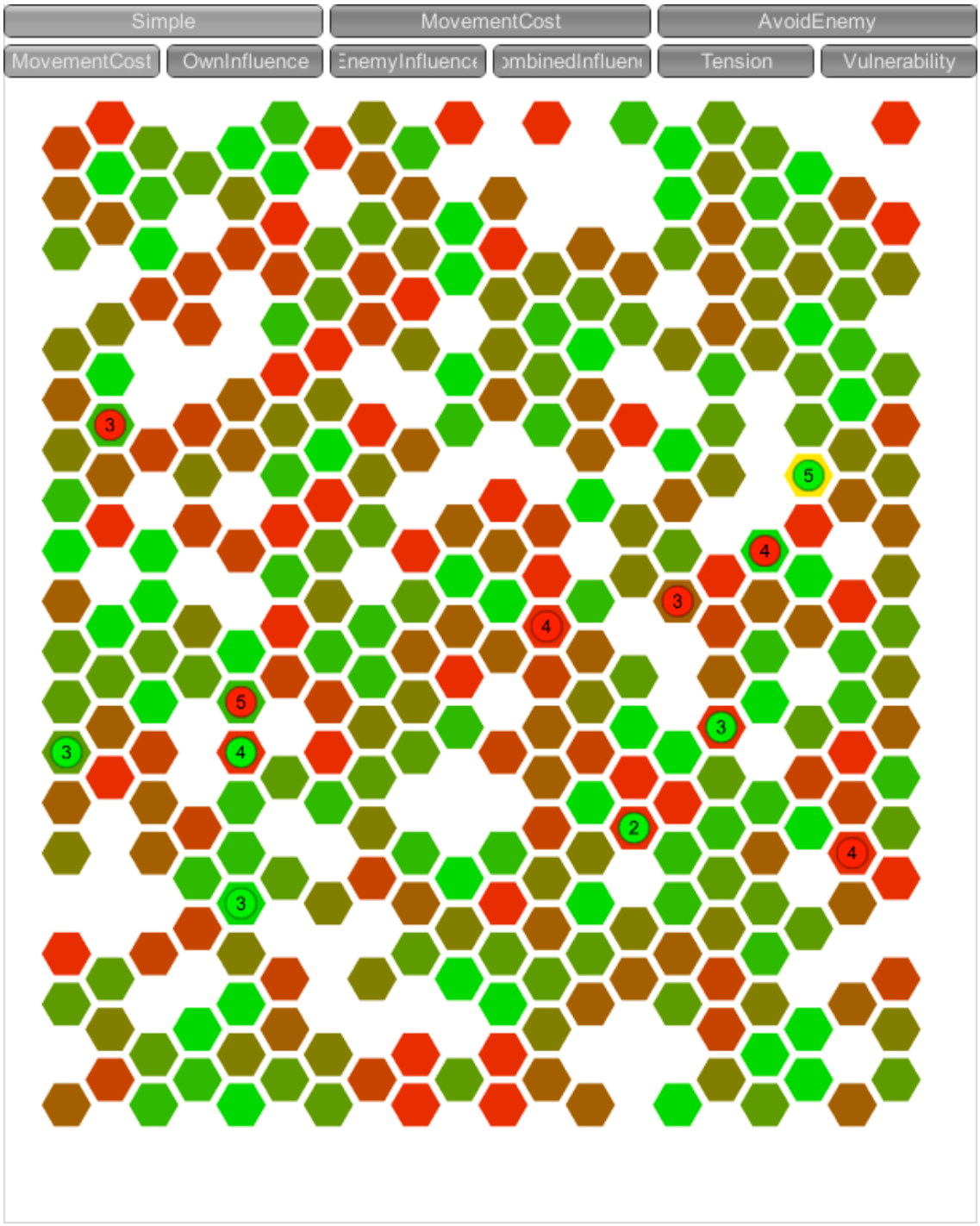
Appendix 1: Screenshots of A* Pathfinder Prototype



Various screenshots from the first version of pathfinder prototype

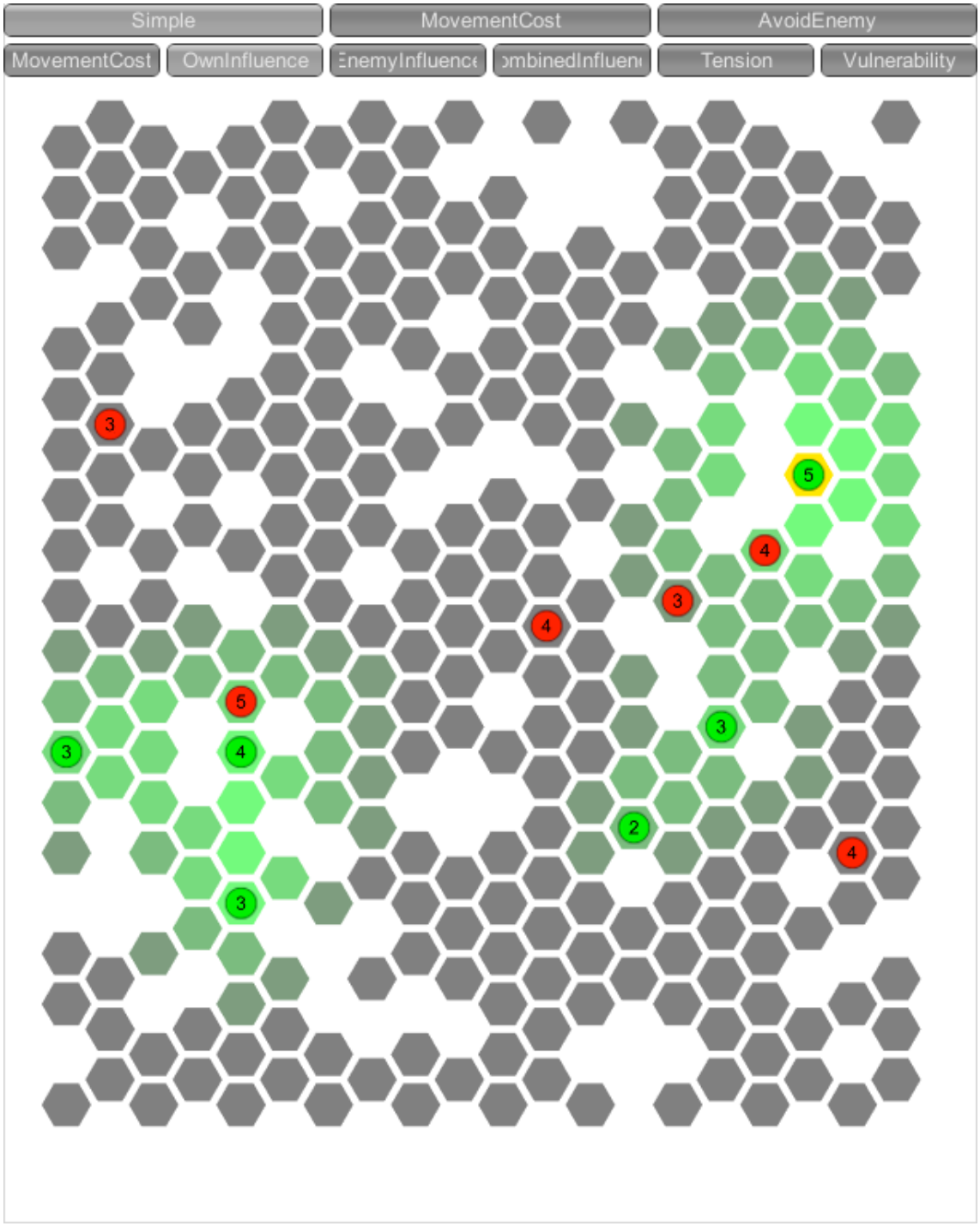
Appendix 2: Screenshots of Spatial Database Prototype Levels of influence

Level 1. Movement cost layer



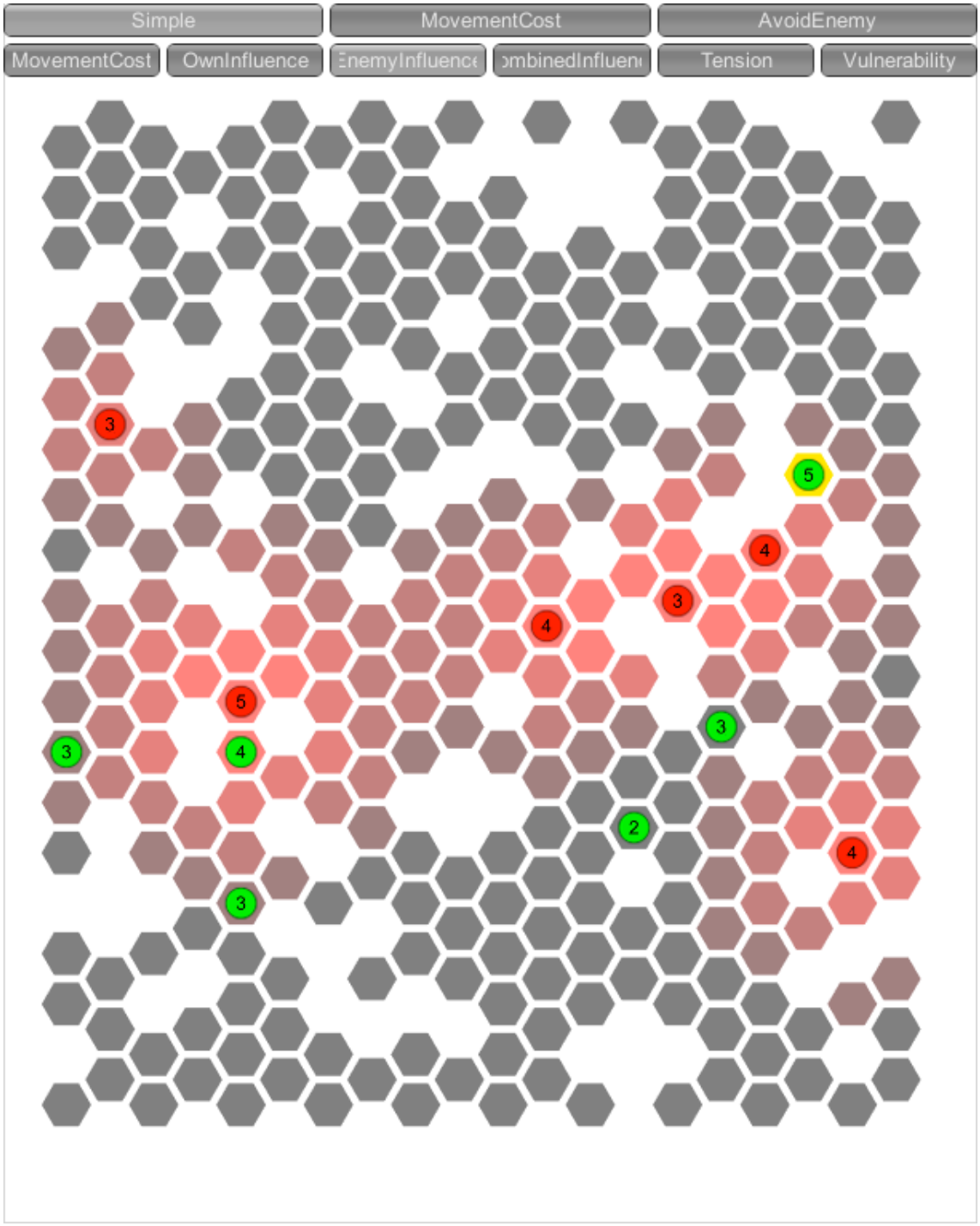
Colors represent movement cost of terrain, ranging from lowest (green) to highest (red) cost.

Level 2. Own influence layer



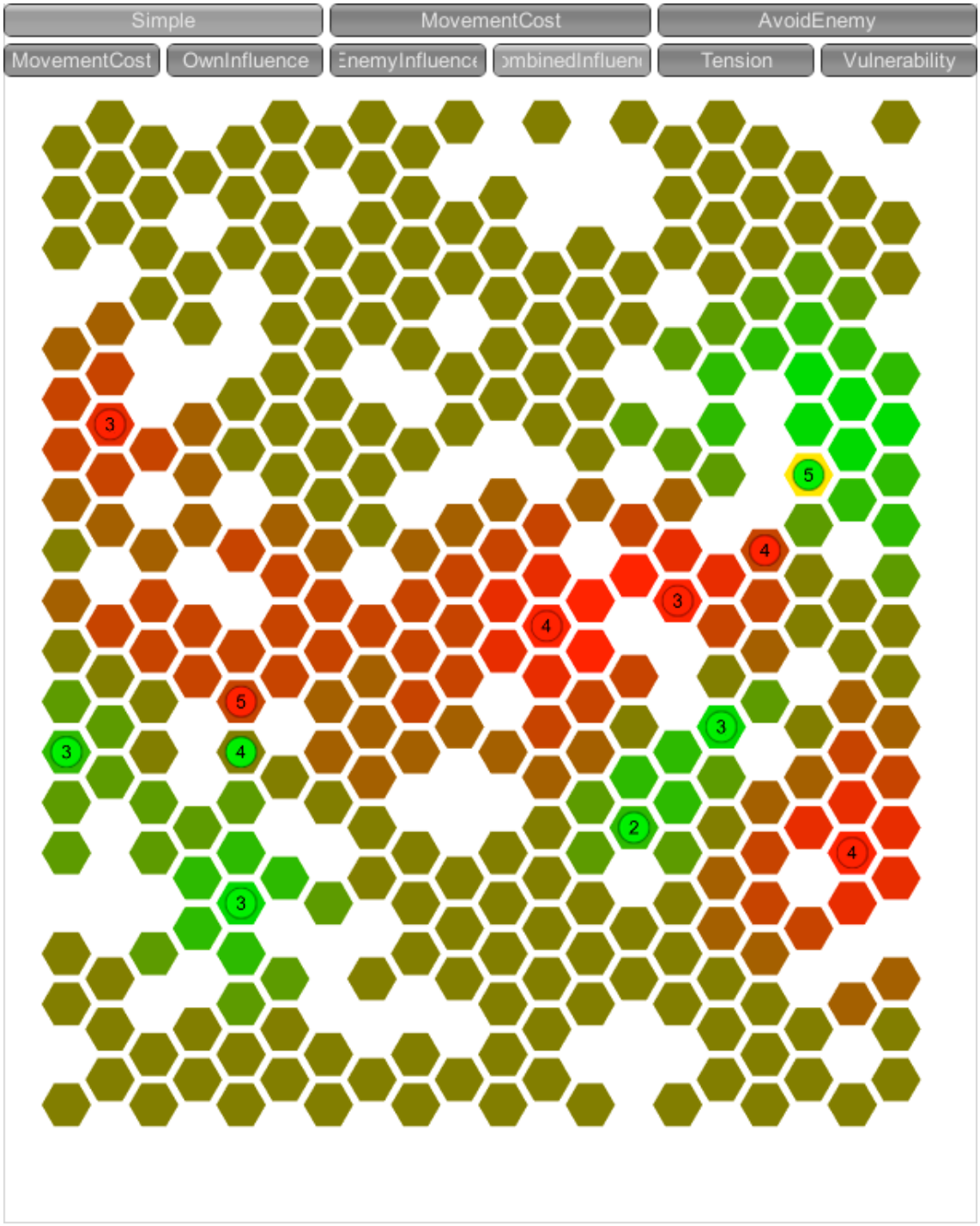
Colors represent player's own influence, ranging from lowest (gray) to highest (light green) influence.

Level 3. Enemy influence layer



Colors represent enemy influence, ranging from lowest (gray) to highest (light red) influence.

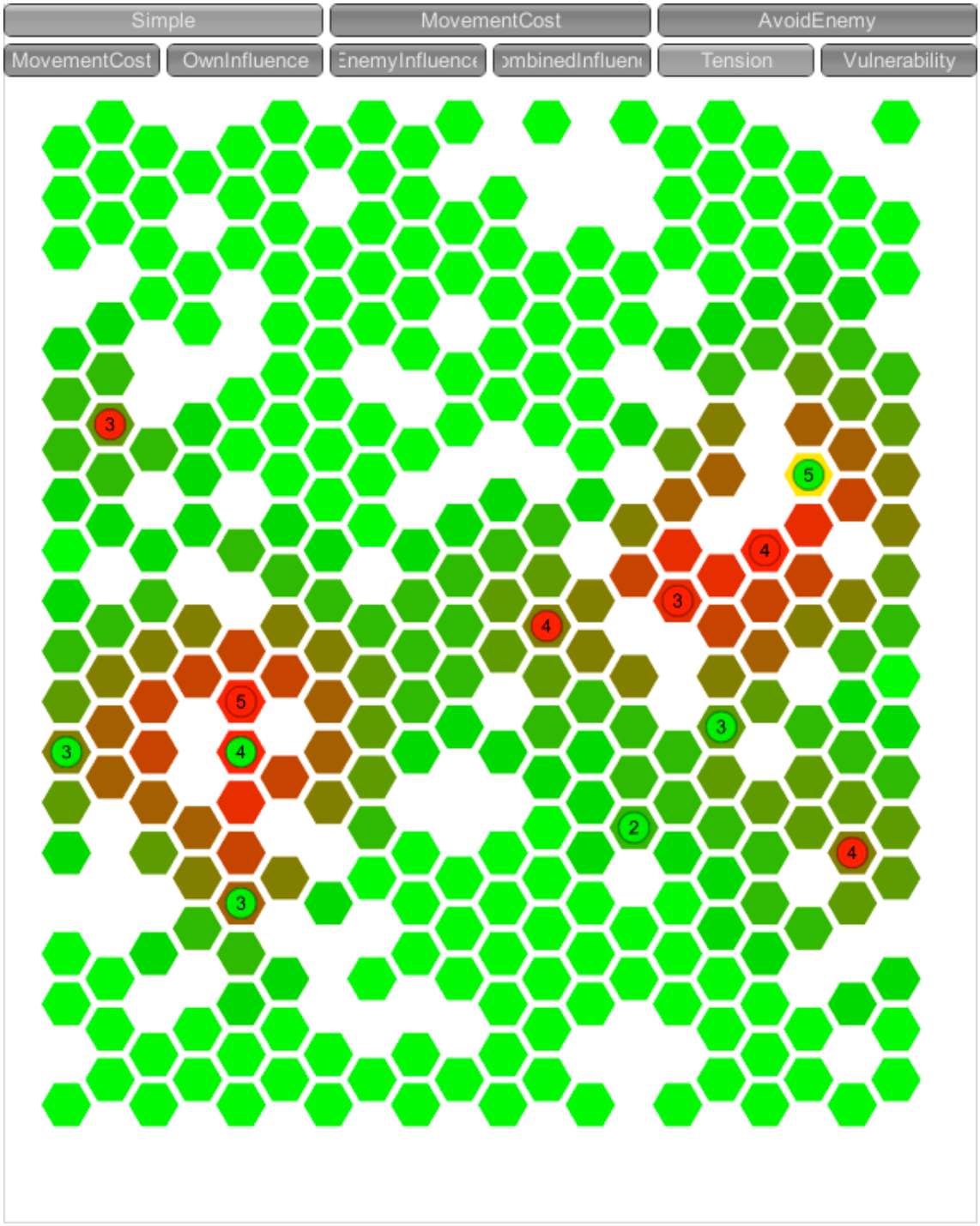
Level 4. Combined influence layer



Combined influence, with green representing own influence and red enemy influence. The data on this layer is calculated using formula:

$$f(\text{combined}) = f(\text{own}) - f(\text{enemy})$$

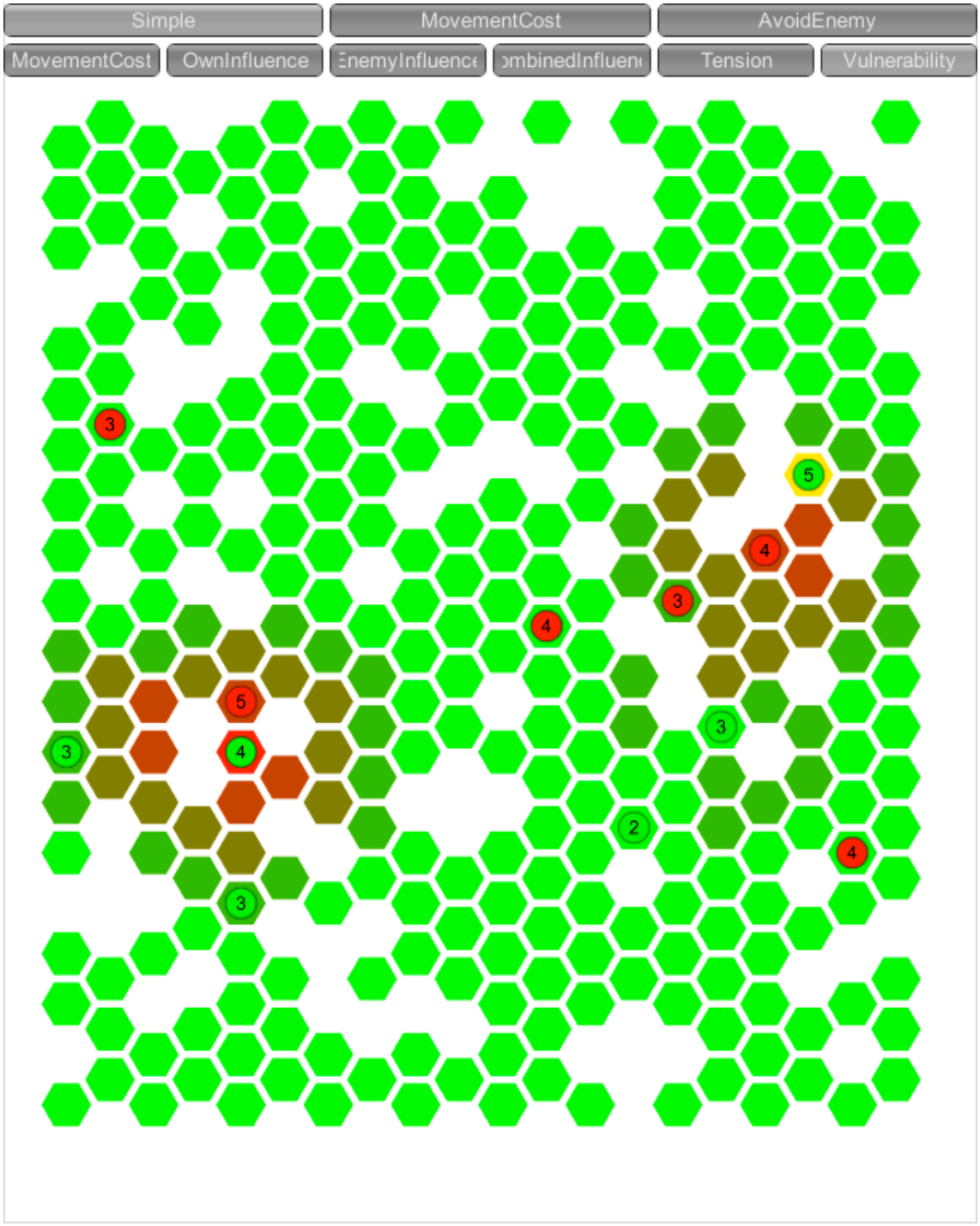
Level 5. Tension layer



Tension layer, representing areas where most military activity is strongest. The data on this layer is calculated using formula:

$$f(\text{tension}) = f(\text{own}) + f(\text{enemy})$$

Level 6. Vulnerability layer

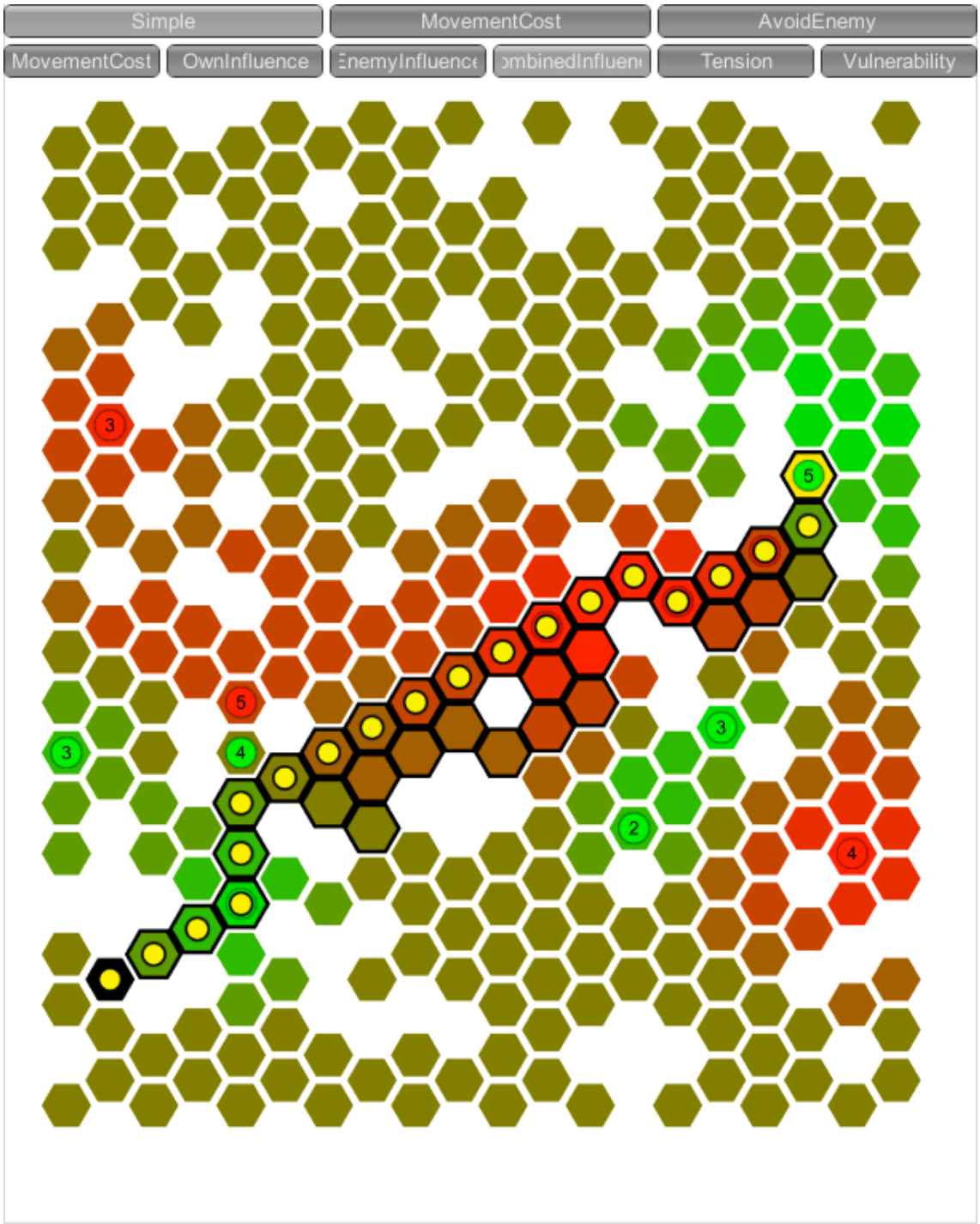


Vulnerability layer, showing areas where either player units facing superior forces they cannot match. The data on this layer is calculated using formula:

$$f(\text{vulnerability}) = f(\text{tension}) - |f(\text{combined})|$$

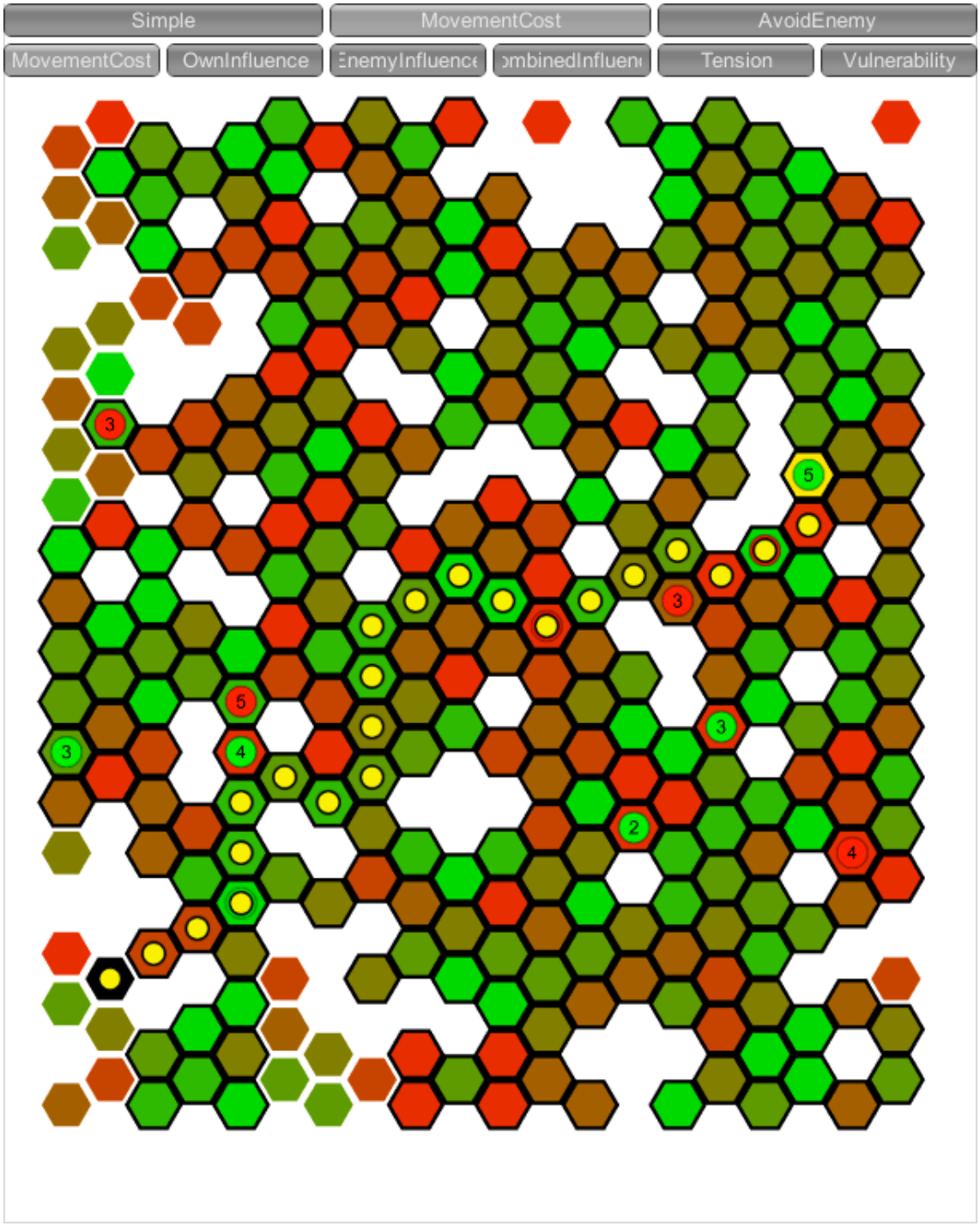
Appendix 3: Screenshots of Tactical Pathfinding Prototype Modes

Pathfinder mode 1. Shortest path selection



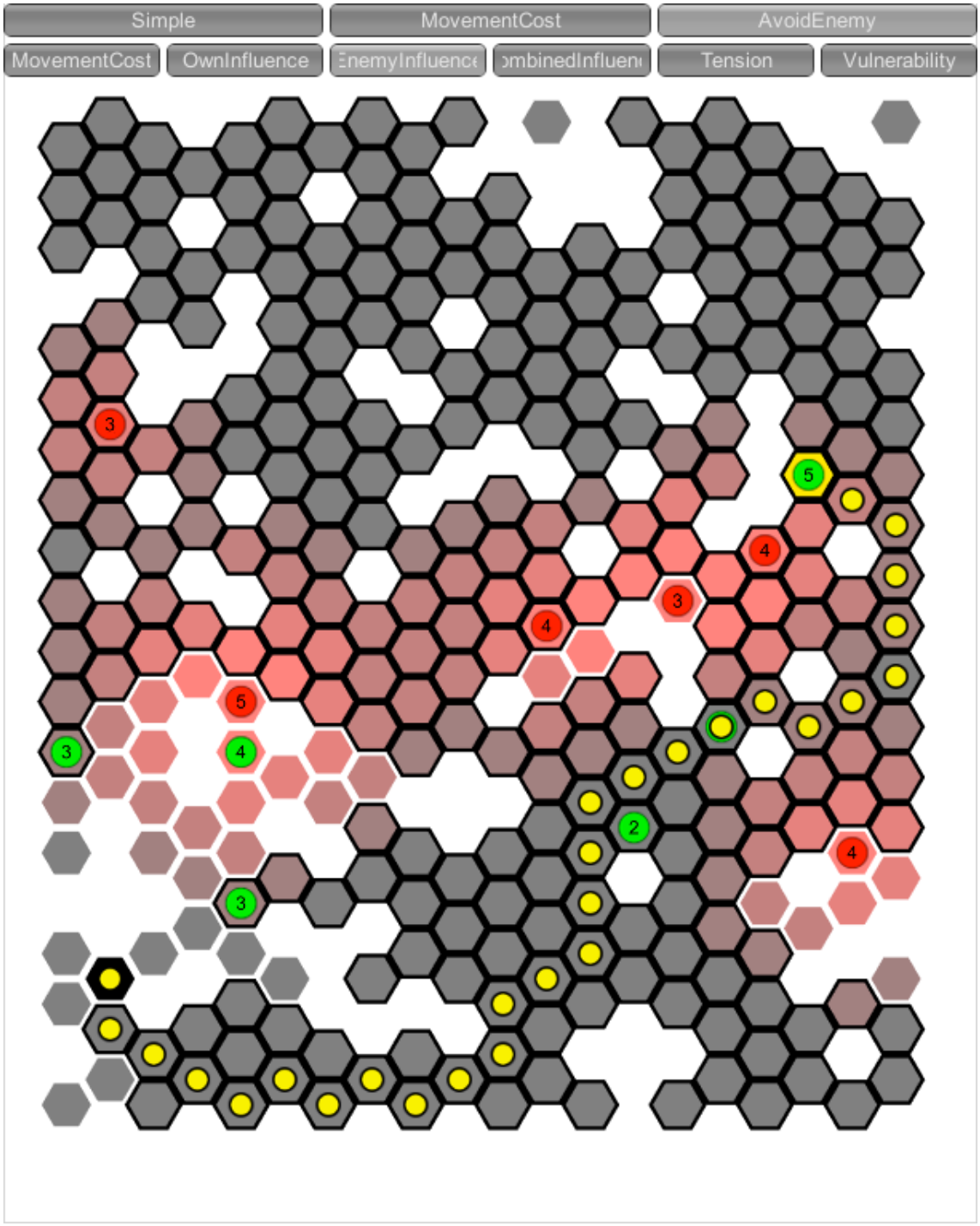
With shortest-path movement, the pathfinder will ignore both terrain type and enemy presence and instead choose the path with lowest number of hex cells. Note that the number of visited nodes (with black borders) is very low thanks to the simple A* heuristic used.

Pathfinder mode 2. Priority based on terrain movement cost



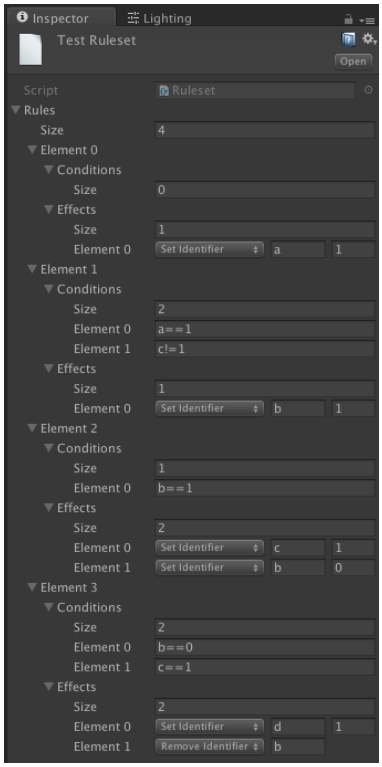
The pathfinder uses the route with lowest movement cost, regardless of enemy presence. Note that the spatial database visualization is set to show terrain movement cost layer.

Pathfinder mode 3. Enemy avoidance priority

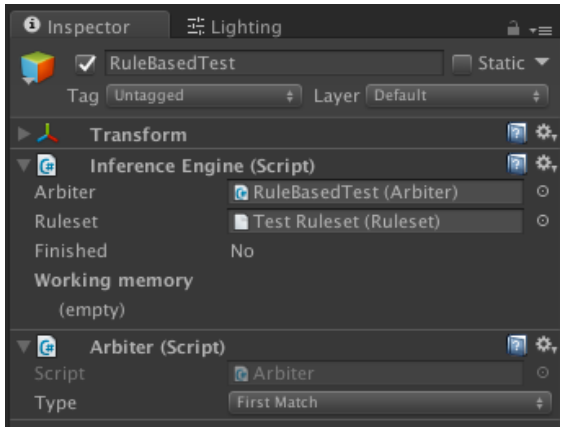


The pathfinder attempts to avoid red areas under enemy influence, favoring the gray nodes with no (or minor) enemy threat. Note that the spatial database visualization is set to show enemy influence.

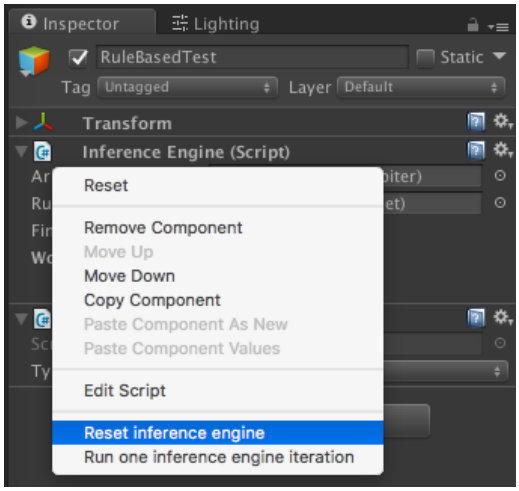
Appendix 4: Screenshots of Inference Engine (Editor Mode) Prototype



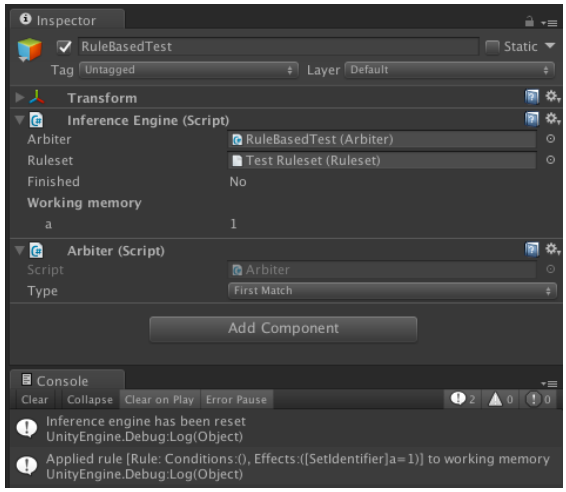
The ruleset asset



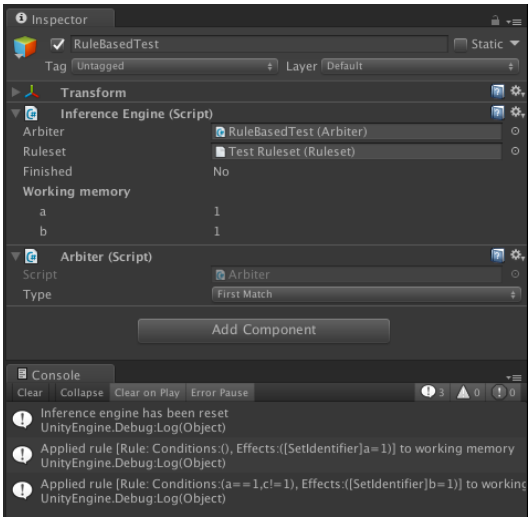
Initial state



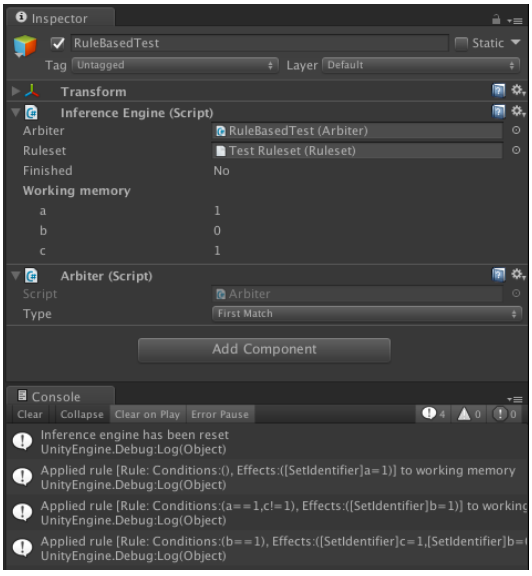
Popup menu with reset and run options



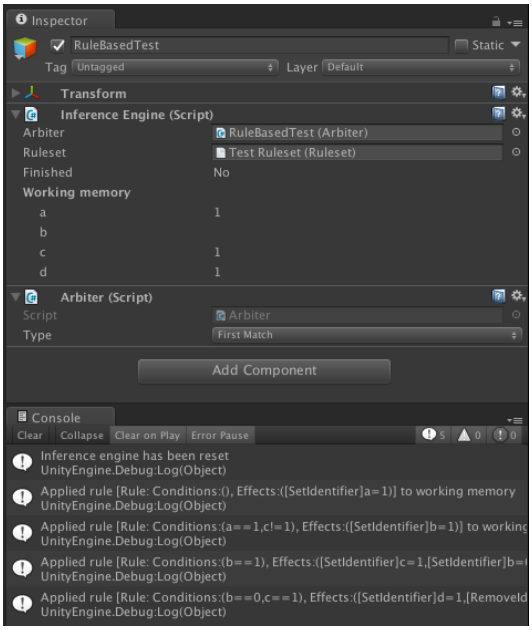
State after first iteration



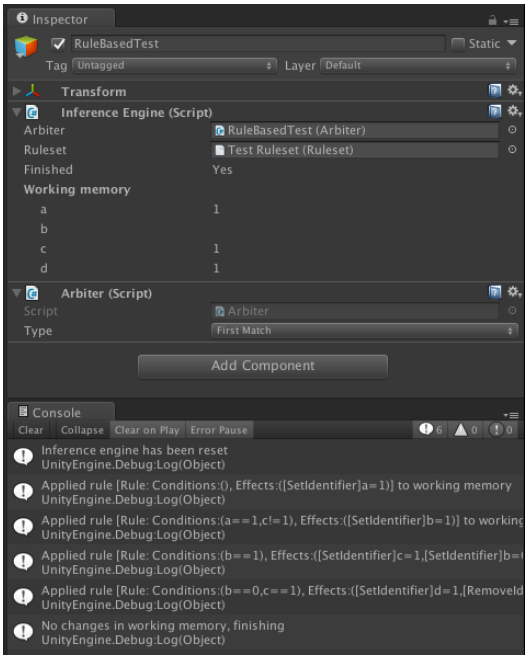
State after second iteration



State after third iteration



State after fourth iteration



Fifth iteration changes no facts, inference process is finished

Appendix 5: Source Code of Pathfinder and Spatial Database Prototype

PathFinder.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;
using System.Linq;

/// <summary>
/// A generic reusable A* pathfinding engine. Uses four interfaces:
/// - IPathFinderSource to provide data about the source graph
/// - IPathFinderNodeSource to provide data for individual nodes
/// - ICostFilter to calculate node traversal costs
/// - IHeuristic for the A* H-cost estimation
///
/// Usage:
///
/// var pathFinder = new PathFinder<MyNodeType>(
///     graphSource, herustic, costFilter);
///
/// pathFinder.Update();
///
/// The HasPath property will indicate if any path was found,
/// and the path data is accessible through the Path property.
/// </summary>
public class PathFinder<CellType,GraphPosition>
    where CellType : IPathFinderNodeSource<CellType,GraphPosition>
    where GraphPosition : System.IEquatable<GraphPosition>
{
    /// <summary>
    /// Result node container for the path query. Used to
    /// track open and closed nodes, and provide the result path.
    /// </summary>
    public class PathFinderNode<CellType2,GraphPosition2>
        where CellType2 : IPathFinderNodeSource<CellType2,GraphPosition2>
        where GraphPosition2 : System.IEquatable<GraphPosition2>
    {
        /// <summary>
        /// Pathfinder which owns this node
        /// </summary>
        PathFinder<CellType2,GraphPosition2> finder;

        /// <summary>
        /// Parent node, from which the search was expanded to this
        /// node. Used to track the path back to start node.
        /// </summary>
        public PathFinderNode<CellType2,GraphPosition2> Parent
            { get; private set; }

        /// <summary>
        /// Original graph node which this internal node maps to
        /// </summary>
        public CellType2 Source { get; private set; }

        /// <summary>
        /// Actual cost of query up to this point. Sum of previous
        /// nodes' and this node's costs.
        /// </summary>
        public float G { get; private set; }
```

```
        /// <summary>
        /// Estimated remaining cost of this node; estimated
        /// distance to goal node.
        /// </summary>
        public float H { get; private set; }

        /// <summary>
        /// Heuristic function F = G + H result, estimated
        /// total path cost when using.
        /// this node.
        /// </summary>
        public float F
        {
            get
            {
                return G + H;
            }
        }

        /// <summary>
        /// Initialize the node container
        /// </summary>
        /// <param name="pf">Reference to the pathfinder which
        /// owns the node</param>
        /// <param name="parent">The parent node, which the query
        /// was expanded from</param>
        /// <param name="source">The actual graph node from user code,
        /// implementing IPathFinderNodeSource interface</param>
        /// <param name="cost">Total cost up to and including
        /// this node</param>
        public PathFinderNode(
            PathFinder<CellType2,GraphPosition2> pf,
            PathFinderNode<CellType2,GraphPosition2> parent,
            CellType2 source,
            float cost)
        {
            this.finder = pf;
            this.Source = source;
            this.Parent = parent;
            this.G = cost;
            this.H = finder.HeuristicFunction.GetHeuristic(
                Source, finder.Source.GetGoal());
        }
    }

    /// <summary>
    /// The query graph source data provider.
    /// </summary>
    public IPathFinderSource<CellType,GraphPosition> Source
        { get; private set; }

    /// <summary>
    /// Resulting path of the query
    /// </summary>
    public List<PathFinderNode<CellType,GraphPosition>> Path
        { get; private set; }

    private Dictionary<GraphPosition,
        PathFinderNode<CellType,GraphPosition>> closedNodes =
        new Dictionary<GraphPosition,
            PathFinderNode<CellType,GraphPosition>>();

    private PriorityQueue<PathFinderNode<CellType,GraphPosition>> openNodes =
```

```
        new PriorityQueue<PathFinderNode<CellType,GraphPosition>>();

    private bool finished;
    private PathFinderNode<CellType,GraphPosition> goalNode;

    /// <summary>
    /// Reference to the heuristic function currently active
    /// </summary>
    public IHeuristic<CellType,GraphPosition> HeuristicFunction { get; set; }

    /// <summary>
    /// The current pathfinding cost filter
    /// </summary>
    public ICostFilter<CellType,GraphPosition> CostFilter { get; set; }

    public bool HasPath
    {
        get
        {
            return Path != null;
        }
    }

    public PathFinder(
        IPathFinderSource<CellType,GraphPosition> source,
        IHeuristic<CellType,GraphPosition> heuristic,
        ICostFilter<CellType,GraphPosition> costFilter)
    {
        this.Source = source;
        this.HeuristicFunction = heuristic;
        this.CostFilter = costFilter;
    }

    /// <summary>
    /// Run the pathfinder query.
    /// </summary>
    public void Update()
    {
        float startTime = Time.realtimeSinceStartup;

        // Clear existing old data
        closedNodes.Clear();
        openNodes.Clear();
        finished = false;
        Path = null;

        // Add starting node to open list
        AddNode(Source.GetStart(), null);

        // Run until either goal found or no more open nodes are available
        while (!openNodes.Empty && !finished)
        {
            // Get highest-priority node from open list
            var node = openNodes.Dequeue();

            // Skip nodes that have already been processed
            if (closedNodes.ContainsKey(node.Source.GetPosition()))
                continue;

            // Add all neighbor nodes that can be traversed
            // into in the open list
            var children = node.Source.GetNeighbors();
            foreach (var child in children)
            {
```

```
                // Skip empty neighbors, happens usually on map edges.
                if (child == null)
                    continue;

                AddNode(child, node);
            }

            // Add this node to closed list
            closedNodes.Add(node.Source.GetPosition(), node);
        }

        // If finished is true, path was found
        if (finished)
        {
            // Trace backwards from goal node to start to build
            // list of result path nodes
            Path = new List<PathFinderNode<CellType,GraphPosition>>();
            var node = goalNode;
            while (node.Parent != null)
            {
                Path.Add(node);
                node = node.Parent;
            }
            Debug.Log("Found path, length: " + Path.Count +
                " cost: " + Path.First().G);
        }
        else
        {
            Debug.Log("No path");
        }

        float endTime = Time.realtimeSinceStartup;

        Debug.Log("Time spent: " + (endTime - startTime).ToString("0.000"));
    }

    private void AddNode(CellType source,
        PathFinderNode<CellType,GraphPosition> parent)
    {
        if (closedNodes.ContainsKey(source.GetPosition()))
            return;

        var parentCost = (parent != null) ? parent.G : 0.0f;

        // Ignore cost for start node
        var thisCost = (parent != null) ? CostFilter.GetCost(source) : 0.0f;

        // Create and insert new node into open list
        var newNode = new PathFinderNode<CellType,GraphPosition>(
                          this,
                          parent,
                          source,
                          thisCost + parentCost);
        openNodes.Insert(newNode.F, newNode);

        // Check if goal reached
        if (source.GetPosition().Equals(Source.GetGoal().GetPosition()))
        {
            finished = true;
            goalNode = newNode;
        }

        openNodes.Validate();
```

```
    }
}
```

IPathFinderSource.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;

/// <summary>
/// Main interface of pathfinder for the map.
/// The generic parameter CellType must be set to the
/// type implementing nodes in the map, and
/// GraphPosition must be the type used to store map
/// coordinates.
/// </summary>
public interface IPathFinderSource<CellType,GraphPosition>
    where CellType : IPathFinderNodeSource<CellType,GraphPosition>
{
    /// <summary>
    /// Gets the start node
    /// </summary>
    CellType GetStart ();

    /// <summary>
    /// Gets the goal node
    /// </summary>
    CellType GetGoal ();

    /// <summary>
    /// Gets the node at a given location on the graph.
    /// </summary>
    CellType GetNodeAt (GraphPosition pos);
}
```

IPathFinderNodeSource.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;

/// <summary>
/// Interface of pathfinder graph nodes for the pathfinder.
/// This should be implemented by the map cell nodes. The generic
/// parameter CellType must be set to the type implementing
/// this interface, and GraphPosition must be the type used
/// to store map coordinates.
/// </summary>
public interface IPathFinderNodeSource<CellType,GraphPosition>
    where CellType : IPathFinderNodeSource<CellType,GraphPosition>
{
    /// <summary>
    /// Get the position of node on graph. Used
    /// also as key for tracking visited nodes.
    /// </summary>
    GraphPosition GetPosition();

    /// <summary>
    /// Get list of neighbor nodes of this node
```

```
    /// in the graph.
    /// </summary>
    CellType[] GetNeighbors();
}
```

ICostFilter.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Cost filter for the pathfinder. The generic
/// parameter CellType must be set to the type
/// implementing nodes in the map, and GraphPosition
/// must be the type used to store map coordinates.
/// </summary>
public interface ICostFilter<CellType,GraphPosition>
    where CellType : IPathFinderNodeSource<CellType,GraphPosition>
{
    /// <summary>
    /// Get pathfinder cost for the given map node.
    /// </summary>
    float GetCost(CellType source);
}
```

IHeuristic.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// Interface for the pathfinder heuristic function.
/// The generic parameter CellType must be set to the
/// type implementing nodes in the map, and
/// GraphPosition must be the type used to store map
/// coordinates.
/// </summary>
public interface IHeuristic<CellType,GraphPosition>
    where CellType : IPathFinderNodeSource<CellType,GraphPosition>
{
    /// <summary>
    /// Calculate the heuristic value for given
    /// source and target nodes.
    /// </summary>
    /// <returns>The estimated heuristic value.</returns>
    /// <param name="source">Source node.</param>
    /// <param name="target">Target node.</param>
    float GetHeuristic(CellType source, CellType target);
}
```

ManhattanDistanceHeuristic.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;
```

```
/// <summary>
/// A simle Manhattan distance heurustic on the hexagonal map grid.
/// </summary>
public class ManhattaDistanceHeurustic<CellType> :
    IHeuristic<CellType,Hex>
    where CellType : IPathFinderNodeSource<CellType,Hex>
{
    public float GetHeuristic(CellType source, CellType target)
    {
        // use the cubic hex distance function as heuristic
        return target.GetPosition().Cube.DistanceTo(
            source.GetPosition().Cube);
    }
}
```

PriorityQueue.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

/// <summary>
/// A very simple priority queue implementation. Uses
/// reversed priority values, where smaller values of
/// priority mean higher priority.
/// </summary>
public class PriorityQueue<T>
{
    /// <summary>
    /// Internal node for priority queue. Basically and single-linked
    /// list with priority(float) and value (V) pair.
    /// </summary>
    private class PQNode<V> where V : T
    {
        public float priority;
        public V value;
        public PQNode<T> next;

        public PQNode(float priority, V value)
        {
            this.priority = priority;
            this.value = value;
        }
    }

    // Root of the linked list
    private PQNode<T> first;

    public bool Empty { get { return first == null; } }

    public void Clear()
    {
        first = null;
    }

    /// <summary>
    /// Insert a value into the priority queue with given priority.
    /// Internally, it iterates the linked list until it finds
    /// the correct nodes between which it should insert it.
    /// </summary>
    /// <param name="priority">Priority of the inserted item
```

```
/// (smaller values indicate higher priority)</param>
/// <param name="value">Actual item being inserted</param>
public void Insert(float priority, T value)
{
    PQNode<T> node = new PQNode<T>(priority, value);

    // List empty? Add as the only node
    if (Empty)
    {
        first = node;
        return;
    }
    else
    {
        var v = first;

        // First item priority lower (greater value) than
        // this node? Add in front and exit.
        if (v.priority >= priority)
        {
            node.next = first;
            first = node;
            return;
        }

        // Loop until proper priority placement found
        for (;;)
        {
            // If ran out of nodes, add as last node
            if (v.next == null)
            {
                v.next = node;
                return;
            }
            // If next node priority is lower,
            // add before it
            else if (v.next.priority >= priority)
            {
                // Insert after current node
                node.next = v.next;
                v.next = node;
                return;
            }
            v = v.next;
        }
    }
}

/// <summary>
/// Validate integrity of the priority queue, checks
/// that no items in queue are out-of-order.
/// </summary>
public void Validate()
{
    var node = first;
    var prevPriority = first.priority;
    while (node.next != null)
    {
        if (node.priority < prevPriority)
            throw new System.FormatException(
                "Priority queue out of order");
        prevPriority = node.priority;
        node = node.next;
    }
```

```
    }

    /// <summary>
    /// Removes and returns the highest-priority value from the queue.
    /// If queue is empty, returns the default value.
    /// </summary>
    public T Dequeue()
    {
        if (Empty)
            return default(T);

        var v = first;
        first = v.next;
        return v.value;
    }
}
```

TestMap.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;
using System.Linq;

/// <summary>
/// Map prototype. Used to test pathfinding and influence maps.
/// </summary>
public class TestMap :
    MonoBehaviour,
    IPathFinderSource<TestMapCell,Hex>,
    IInfluenceMap<TestUnit,TestMap.SpatialDataLayer>
{
    /// <summary>
    /// Layers used for spatial database
    /// </summary>
    public enum SpatialDataLayer
    {
        MovementCost,
        OwnInfluence,
        EnemyInfluence,
        CombinedInfluence,
        Tension,
        Vulnerability
    }

    /// <summary>
    /// Different modes for the pathfinder
    /// </summary>
    public enum PathfinderMode
    {
        Simple,
        MovementCost,
        AvoidEnemy
    }

    /// <summary>
    /// Map width
    /// </summary>
    public const int Width = 20;

    /// <summary>
```

```
/// Map height
/// </summary>
public const int Height = 20;

// MonoBehaviour fields exposed to Unity editor

public GameObject mapCellPrefab;
public GameObject mapUnitPrefab;
public Transform mapRoot;
public Transform unitsRoot;
public Camera cameraRef;
public int numUnits = 6;
public PathfinderMode pathfinderMode;

public TestUnit ActiveUnit { get; private set; }
public SpatialDatabase<TestMap,TestUnit,SpatialDataLayer>
    SpatialDatabase { get; private set; }
public PathFinder<TestMapCell,Hex> PathFinder { get; private set; }

private TestMapCell[,] map;
private Dictionary<int,TestUnit> units = new Dictionary<int, TestUnit>();
private Hex? currentTarget;
private Hex? currentStart;

/// <summary>
/// Test cost filter, used by pathfinder to query graph
/// traversal costs for given nodes.
/// </summary>
private class TestCostFilter : ICostFilter<TestMapCell,Hex>
{
    public TestMap map;

    /// <summary>
    /// Just passes the query to TestMap.GetPathfinderCost method
    /// </summary>
    public float GetCost(TestMapCell source)
    {
        return map.GetPathfinderCost(source.GetPosition());
    }
}

public void Start()
{
    // Create the spatial database for this map
    SpatialDatabase =
        new SpatialDatabase<TestMap,TestUnit,SpatialDataLayer>(
        this, Width, Height);

    // Create map
    map = new TestMapCell[Width, Height];
    for (int y = 0; y < Height; y++)
        for (int x = 0; x < Width; x++)
        {
            // Convert offset coordinate to hex position
            var hexPos = new Hex (new OffsetPos (x, y));

            // Instantiate map cell prefab on the map
            var cellGO = Instantiate(mapCellPrefab);
            cellGO.transform.SetParent(mapRoot);
            var cell = cellGO.GetComponent<TestMapCell>();

            // Initialize the map cell with 1/6 (≈16%) chance
            // of being obstacle
            cell.Init(this, hexPos,
```

```
                    (Random.Range(1,6) == 1) ? TestMapCell.NodeType.Obstacle :
                    TestMapCell.NodeType.Plain);

                // Update spatial database movement cost layer with random
                // cost between 1..8
                SpatialDatabase.SetData(x, y,
                    (float)Random.Range(1, 8),
                    SpatialDataLayer.MovementCost);

                map[x, y] = cell;
            }

        // Create units
        int id = 0;
        for (int i = 0; i < numUnits; i++)
        {
            // p = 0 for friendly, p = 1 for enemy
            for (int p = 0; p < 2; p++)
            {
                // Pick random position which is not obstacle,
                // and does not contain any unit yet
                var pos = GetRandomPos((n) =>
                    n.Type == TestMapCell.NodeType.Plain &&
                    GetUnitAt(n.GetPosition()) == null);

                // Instantiate unit prefab on the map
                var unitGO = Instantiate(mapUnitPrefab);
                unitGO.transform.SetParent(unitsRoot);
                var unit = unitGO.GetComponent<TestUnit>();

                // Initialize the unit with random strength in range 2..6
                // and set friendly flag
                unit.Init(this, pos, id, Random.Range(2,6), p == 0);

                units[id++] = unit;
            }
        }

        // Update spatial database
        SpatialDatabase.RecalculateInfluence();

        // Create pathfinder instance
        PathFinder = new PathFinder<TestMapCell,Hex>(
            this,
            new ManhattaDistanceHeurustic<TestMapCell>(),
            new TestCostFilter { map = this });

        FixCamera();

        // Set random unit as active unit, and pick random goal
        SetActiveUnit(GetRandomUnit((u) => u.Friendly));
        SetTarget(GetRandomPos((n) => n.Type == TestMapCell.NodeType.Plain));
    }

    /// <summary>
    /// Get random position which matches the given filter.
    /// </summary>
    /// <returns>The random position.</returns>
    /// <param name="filter">Filter to match map hexes with.</param>
    private Hex GetRandomPos(System.Func<TestMapCell,bool> filter)
    {
        Hex pos;
        // Loop until a location is picked which passes the filter
        do
```

```
    {
        pos = new Hex(new OffsetPos(
            Random.Range(0, Width - 1),
            Random.Range(0, Height - 1)));
    }
    while (!filter(GetNodeAt(pos)));

    return pos;
}

/// <summary>
/// Get random unit which matches the given filter.
/// </summary>
/// <returns>The randomly picked unit, null if no
/// units exist for the given filter.</returns>
/// <param name="filter">Filter to match units with.</param>
private TestUnit GetRandomUnit(System.Func<TestUnit,bool> filter)
{
    // Use LINQ OrderBy with Random to randomize order
    var rnd = new System.Random();
    return units.Values.
        Where(u => filter(u)).
        OrderBy(u => rnd.Next()).
        FirstOrDefault();
}

/// <summary>
/// Set the active unit, updates pathfinder start
/// position and clears old route
/// </summary>
/// <param name="unit">Unit to select.</param>
private void SetActiveUnit(TestUnit unit)
{
    ActiveUnit = unit;
    SetStart(unit.Pos);
    ClearRoute();
}

/// <summary>
/// Gets the unit at given location
/// </summary>
/// <returns>Reference to the unit at this location,
/// null if no unit here.</returns>
/// <param name="pos">Position on map.</param>
public TestUnit GetUnitAt(Hex pos)
{
    return units.Values.FirstOrDefault(u => u.Pos == pos);
}

/// <summary>
/// Gets the map node at given location
/// </summary>
/// <returns>Reference to the map cell at this location,
/// null if out of range.</returns>
/// <param name="pos">Position on map.</param>
public TestMapCell GetNodeAt (Hex pos)
{
    var offs = pos.Offset;
    if (!offs.InRange(0, 0, Width - 1, Height - 1))
        return null;
    return map[offs.X, offs.Y];
}

/// <summary>
```

```
/// Used to provide the start node to pathfinder
/// </summary>
/// <returns>The start node.</returns>
public TestMapCell GetStart ()
{
    return this.GetNodeAt(currentStart.Value);
}

/// <summary>
/// Used to provide the goal node to pathfinder
/// </summary>
/// <returns>The goal node.</returns>
public TestMapCell GetGoal ()
{
    return this.GetNodeAt(currentTarget.Value);
}

/// <summary>
/// Gets the pathfinder cost for given hex cell.
/// Uses the selected pathfinder mode to choose
/// the appropriate source(s) and returns the
/// resulting cost.
/// </summary>
/// <returns>The cost of given hex.</returns>
/// <param name="pos">Position on map to query.</param>
public float GetPathfinderCost(Hex pos)
{
    int x = pos.Offset.X, y = pos.Offset.Y;
    switch (pathfinderMode)
    {
        case PathfinderMode.Simple:
            // For simple cost always return 1 for each node
            return 1;
        case PathfinderMode.MovementCost:
            // Return the terrain movement cost value
            return SpatialDatabase.GetData(x, y,
                SpatialDataLayer.MovementCost);
        case PathfinderMode.AvoidEnemy:
            // When avoiding enemy, use 10x score from enemy influence,
            // but add 1x movement cost to allow units to optimize
            // terrain movement when no enemies nearby
            return 10 *
                SpatialDatabase.GetData(x, y,
                    SpatialDataLayer.EnemyInfluence) +
                SpatialDatabase.GetData(x, y,
                    SpatialDataLayer.MovementCost);
    }
    return 1;
}

/// <summary>
/// Set pathfinder start position
/// </summary>
/// <param name="start">Start location</param>
public void SetStart(Hex start)
{
    if (currentStart.HasValue)
    {
        var oldOffs = currentStart.Value.Offset;
        map[oldOffs.X, oldOffs.Y].SetType(TestMapCell.NodeType.Plain);
    }
    var offs = start.Offset;
    map[offs.X, offs.Y].SetType(TestMapCell.NodeType.Start);
    currentStart = start;
```

```
}

/// <summary>
/// Handles user input (mouse click) on the map.
/// When clicking friendly unit, make it the new
/// pathfinding source, otherwise set the clicked
/// hex as pathfinder goal.
/// </summary>
/// <param name="pos">Position clicked.</param>
public void ClickMap(Hex pos)
{
    var unit = GetUnitAt(pos);
    if (unit != null && unit.Friendly)
    {
        SetActiveUnit(unit);
    }
    else
    {
        SetTarget(pos);
    }
}

/// <summary>
/// Clear the old pathfinder route visualization
/// and target.
/// </summary>
private void ClearRoute()
{
    for (int y = 0; y < Height; y++)
        for (int x = 0; x < Width; x++)
        {
            map[x, y].Spot = false;
            map[x, y].SetType(map[x, y].Type);
            map[x, y].Visited = false;
        }

    if (currentTarget.HasValue)
    {
        var oldOffs = currentTarget.Value.Offset;
        map[oldOffs.X, oldOffs.Y].SetType(TestMapCell.NodeType.Plain);
        currentTarget = null;
    }
}

/// <summary>
/// Force map refresh, updates the hex cell colors to
/// match currently active spatial database layer.
/// </summary>
public void RefreshMapColors()
{
    for (int y = 0; y < Height; y++)
        for (int x = 0; x < Width; x++)
        {
            map[x, y].Refresh();
        }
}

/// <summary>
/// Set new pathfinder target. Clears the old path, and
/// triggers new pathfinder query, updating the map
/// spots if path was found.
/// </summary>
/// <param name="target">Target.</param>
public void SetTarget(Hex target)
```

```
{
    ClearRoute();

    var offs = target.Offset;
    var cell = map[offs.X, offs.Y];
    if (cell.Type == TestMapCell.NodeType.Plain)
    {
        cell.SetType(TestMapCell.NodeType.Goal);
        currentTarget = target;
    }

    if (currentTarget.HasValue)
    {
        PathFinder.Update();
        if (PathFinder.HasPath)
        {
            foreach (var pfNode in PathFinder.Path)
            {
                var node = pfNode.Source;
                node.Spot = true;
            }
        }
    }
}

/// <summary>
/// Refresh the pathfinder path. Basically
/// just sets the target to its current value
/// to trigger pathfinder update.
/// </summary>
private void UpdatePath()
{
    if (currentTarget.HasValue)
    {
        SetTarget(currentTarget.Value);
    }
}

/// <summary>
/// Resize and position camera to fit map in the
/// orthogonal viewport
/// </summary>
private void FixCamera()
{
    var bottomLeftHex = new Hex(new OffsetPos(0, 0));
    var topRightHex = new Hex(new OffsetPos(Width, Height));

    var bottomLeft = bottomLeftHex.GetCenter(0.5f);
    var topRight = topRightHex.GetCenter(0.5f);

    var center = (bottomLeft + topRight) / 2.0f;
    center.y -= Hex.Sqrt3 * 0.25f * 0.5f;
    center.x -= Hex.Sqrt3 * 0.25f * 0.5f;

    cameraRef.transform.position =
        new Vector3(center.x, center.y, cameraRef.transform.position.z);
    cameraRef.orthographicSize =
        Mathf.Abs(topRight.x - bottomLeft.x + Hex.Sqrt3) *
        0.5f / cameraRef.aspect;
}

/// <summary>
/// Gets the influence sources. Used by Spatial Database.
/// </summary>
```

```
/// <returns>The influence sources which affect the given layer.</returns>
/// <param name="layer">Layer which source affect</param>
public IEnumerable<TestUnit> GetInfluenceSources(SpatialDataLayer layer)
{
    if (layer == SpatialDataLayer.EnemyInfluence)
    {
        // Enemy influence is affected by enemy unit strengths
        return units.Values.Where(u => !u.Friendly);
    }
    else if (layer == SpatialDataLayer.OwnInfluence)
    {
        // Own influence is affected by friendly unit strengths
        return units.Values.Where(u => u.Friendly);
    }
    else
        return null;
}

/// <summary>
/// Propagates the influence of given source to this layer
/// </summary>
/// <param name="layer">Layer to output influence on.</param>
/// <param name="source">The influence source.</param>
public void PropagateInfluence(SpatialDataLayer layer, TestUnit source)
{
    int strength = source.Strength;

    // Add unit location influence first
    SpatialDatabase.AddData(
        source.Pos.Offset.X, source.Pos.Offset.Y, strength, layer);

    // Spread influence linearly, reducing effect by one for
    // each step further from the source unit. Each iteration
    // applies the influence as a hexagonal ring around the
    // source.
    for (int i = 1; i <= strength; i++)
    {
        // Apply to each hex direction
        foreach (var dir in Hex.directions)
        {
            // Get one edge of this hexagonal circle to start
            // from. Loop for the number of hexes
            // on this particular side to apply to.
            var rimPos = source.Pos.Cube + dir.Value * i;
            for (int r = 0; r < i; r++)
            {
                // The ring direction is two steps clockwise from
                // the edge direction:
                //
                //   P-_        O: unit location
                //   |  -T      P: circle corner (rimPos)
                //   |          T: next corner (2 steps in clockwise
                //   O             direction) to iterate towards
                //
                // Each iteration of [r] steps from P towards T
                rimPos += Hex.directions[
                    (Hex.Side)(((int)dir.Key + 2) %
                    (int)Hex.Side.NumSides)];
                var rimHex = new Hex(rimPos);
                // Check range in case we hit edge of map
                if (rimHex.Offset.InRange(0, 0, Width - 1, Height - 1))
                {
                    SpatialDatabase.AddData(
                        rimHex.Offset.X, rimHex.Offset.Y,
```

```
                        strength - i, layer);
                }
            }
        }
    }
}

/// <summary>
/// Layer influence filter. Used to combine lower layers into
/// higher-level layers.
/// </summary>
/// <returns>The output value of influence for this cell.</returns>
/// <param name="layer">Layer being filtered.</param>
/// <param name="x">The x coordinate on map.</param>
/// <param name="y">The y coordinate on map.</param>
public float FilterLayerInfluence(SpatialDataLayer layer, int x, int y)
{
    switch (layer)
    {
        case SpatialDataLayer.CombinedInfluence:
            // combinedInfluence = ownInfluence - enemyInfluence
            {
                return SpatialDatabase.GetData(x, y,
                    SpatialDataLayer.OwnInfluence) -
                    SpatialDatabase.GetData(x, y,
                        SpatialDataLayer.EnemyInfluence);
            }
        case SpatialDataLayer.Tension:
            // tensionLevel = ownInfluence + enemyInfluence
            {
                return SpatialDatabase.GetData(x, y,
                    SpatialDataLayer.OwnInfluence) +
                    SpatialDatabase.GetData(x, y,
                        SpatialDataLayer.EnemyInfluence);
            }
        case SpatialDataLayer.Vulnerability:
            // vulnerabilityLevel = tensionLevel - Abs(combinedInfluence)
            {
                return SpatialDatabase.GetData(x, y,
                    SpatialDataLayer.Tension) -
                    Mathf.Abs(SpatialDatabase.GetData(x, y,
                        SpatialDataLayer.CombinedInfluence));
            }
    }
    return 0.0f;
}

/// <summary>
/// Show the debug buttons on top of the game view
/// </summary>
void OnGUI()
{
    // Button grid for selecting active pathfinder mode
    var newPathfinderMode = (PathfinderMode)GUILayout.SelectionGrid(
        (int)pathfinderMode,
        System.Enum.GetNames(typeof(PathfinderMode)),
        System.Enum.GetValues(typeof(PathfinderMode)).Length);
    if (newPathfinderMode != pathfinderMode)
    {
        pathfinderMode = newPathfinderMode;
        UpdatePath();
    }

    // Button grid for selecting which spatial database layer
```

```
        // is being visualized on map
        var newActiveLayer = (SpatialDataLayer)GUILayout.SelectionGrid(
            (int)SpatialDatabase.ActiveLayer,
            System.Enum.GetNames(typeof(SpatialDataLayer)),
            System.Enum.GetValues(typeof(SpatialDataLayer)).Length);
        if (newActiveLayer != SpatialDatabase.ActiveLayer)
        {
            SpatialDatabase.ActiveLayer = newActiveLayer;
            RefreshMapColors();
        }
    }
}
```

TestMapCell.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using Utilities.Hex;

/// <summary>
/// Cell for the prototype map.
/// </summary>
public class TestMapCell :
    MonoBehaviour,
    IPathFinderNodeSource<TestMapCell,Hex>
{
    /// <summary>
    /// The type of this map cell
    /// </summary>
    public enum NodeType
    {
        Plain,
        Obstacle,
        Start,
        Goal
    }

    private TestMap map;

    /// <summary>
    /// Position of this cell on the map
    /// </summary>
    private Hex pos;

    // MonoBehaviour Fields exposed to Unity Editor
    public SpriteRenderer cellSprite;
    public SpriteRenderer spotSprite;
    public SpriteRenderer borderSprite;

    public NodeType Type { get; private set; }

    private bool _spot = false;
    /// <summary>
    /// Toggle the spot visualization, used to visualize
    /// the pathfinder result on the map.
    /// </summary>
    public bool Spot
    {
        get { return _spot; }
        set { _spot = value; UpdateSpot(); }
    }
```

```
private bool _visited = false;
/// <summary>
/// The visited flag, used to visualize nodes
/// that were visited during pathfinding.
/// </summary>
public bool Visited
{
    get { return _visited; }
    set { _visited = value; UpdateBorders(); }
}

public void Init(TestMap map, Hex pos, NodeType type)
{
    this.map = map;
    this.pos = pos;

    // Set physical world position of this cell
    // based on the hex location
    Vector2 v2 = pos.GetCenter(0.5f);
    transform.position = new Vector3(v2.x, v2.y, 0.0f);

    SetType(type);
    UpdateSpot();
}

/// <summary>
/// Handle user input when this map cell is clicked.
/// </summary>
public void OnMouseDown()
{
    map.ClickMap(pos);
}

/// <summary>
/// Update the node type. Also updates spatial database
/// visualization.
/// </summary>
/// <param name="type">New type of the node</param>
public void SetType(NodeType type)
{
    Type = type;
    switch (type)
    {
        case NodeType.Goal:
            // Goal node, always black
            {
                cellSprite.color = Color.black;
                break;
            }
        case NodeType.Obstacle:
            // Obstacle node, always white
            {
                cellSprite.color = Color.white;
                break;
            }
        case NodeType.Plain:
            // Plain node
            {
                var value = map.SpatialDatabase.GetData(
                    pos.Offset.X, pos.Offset.Y);
                switch (map.SpatialDatabase.ActiveLayer)
                {
```

```
                    case TestMap.SpatialDataLayer.MovementCost:
                        // Movement colors range from
                        // green (lowest) to red (highest)
                        {
                            value /= 8.0f;
                            cellSprite.color =
                                new Color(value, 1.0f - value, 0.0f);
                            break;
                        }
                    case TestMap.SpatialDataLayer.OwnInfluence:
                        // Own influence colors range from
                        // gray (lowest) to green (highest)
                        {
                            value /= 4.0f;
                            cellSprite.color = new Color(
                                0.5f, 0.5f + value / 2.0f, 0.5f);
                            break;
                        }
                    case TestMap.SpatialDataLayer.EnemyInfluence:
                        // Enemy influence colors range from
                        // gray (lowest) to red (highest)
                        {
                            value /= 4.0f;
                            cellSprite.color = new Color(
                                0.5f + value / 2.0f, 0.5f, 0.5f);
                            break;
                        }
                    case TestMap.SpatialDataLayer.CombinedInfluence:
                        // Combined influence colors range from
                        // red (enemy) to green (own)
                        {
                            value /= 4.0f;
                            cellSprite.color = new Color(
                                0.5f - value * 0.5f,
                                0.5f + value * 0.5f,
                                0.0f);
                            break;
                        }
                    case TestMap.SpatialDataLayer.Tension:
                        // Tension colors range from
                        // green (low) to red (high)
                        {
                            value /= 4.0f;
                            cellSprite.color = new Color(
                                value * 0.5f, 1.0f - value * 0.5f, 0.0f);
                            break;
                        }
                    case TestMap.SpatialDataLayer.Vulnerability:
                        // Vulnerability colors range from
                        // green (low) to red (high)
                        {
                            value /= 4.0f;
                            cellSprite.color = new Color(
                                value * 0.5f, 1.0f - value * 0.5f, 0.0f);
                            break;
                        }
                }
                break;
            }
        case NodeType.Start:
            // Start node, always yellow
            {
                cellSprite.color = Color.yellow;
                break;
```

```
                }
            }
        }

        /// <summary>
        /// Refresh the visualization of this node
        /// </summary>
        public void Refresh()
        {
            SetType(Type);
        }

        private void UpdateSpot()
        {
            spotSprite.gameObject.SetActive(_spot);
        }

        private void UpdateBorders()
        {
            borderSprite.color = _visited ? Color.black : Color.white;
        }

        /// <summary>
        /// Returns this node's position to the pathfinder
        /// </summary>
        public Hex GetPosition()
        {
            return pos;
        }

        /// <summary>
        /// Get list of neighbor nodes of this for the pathfinder
        /// </summary>
        public TestMapCell[] GetNeighbors()
        {
            Visited = true;

            int i = 0;
            var result = new TestMapCell[(int)Hex.Side.NumSides];
            foreach (var side in Hex.directions.Keys)
            {
                var dirPos = Hex.directions[side];
                var newPos = pos + new Hex(dirPos);
                var node = map.GetNodeAt(newPos);
                if (node != null && node.Type != NodeType.Obstacle)
                {
                    result[i++] = node;
                }
            }
            return result;
        }
    }
```

Hex.cs

```
using System;
using System.Collections.Generic;
using UnityEngine;

namespace Utilities.Hex
{
```

```
/// <summary>
/// Functionality for handling hex grid coordinates
///
/// This struct represents a single coordinate in
/// hexagonal grid space, with three
/// possible presentations of its state:
///
/// Cube:
///
///     XYZ         ->          0,1,-1
/// XYZ     XYZ     ->  -1,1,0          1,0,-1
///     XYZ         ->          0,0,0
/// XYZ     XYZ     ->  -1,0,1          1,-1,0
///     XYZ         ->          0,-1,1
///
/// Axial:
///
/// XY            ->  0,0
///    XY         ->      1,0
/// XY    XY     ->  0,1     2,0
///    XY    XY     ->          1,1     3,0
/// XY    XY     ->    0,2     2,1
///
/// Offset:
///
/// XY    XY     ->  0,0     2,0
///    XY    XY     ->      1,0     3,0
/// XY    XY     ->  0,1     2,1
///    XY    XY     ->          1,1     3,1
///
/// The canical coordinate is kept in memory as axial
/// coordinate to save memory, calculations are
/// performed in cube space, and offset coordinates can be used
/// to optimize storage of map data which usually is presented
/// as interleaved hex grid.
/// </summary>
public struct Hex
{
    public enum Side
    {
        TopLeft = 0,
        Top = 1,
        TopRight = 2,
        BottomRight = 3,
        Bottom = 4,
        BottomLeft = 5,
        NumSides = 6
    }

    /// <summary>
    /// Shortcuts for the six neighbors of hex coordinates
    /// </summary>
    static public readonly Dictionary<Side,CubicPos> directions =
        new Dictionary<Side,CubicPos>()
    {
        {
            Side.TopLeft,
            new CubicPos(-1, 1, 0)
        },
        {
            Side.Top,
            new CubicPos(0, 1, -1)
        },
        {
```

```
            Side.TopRight,
            new CubicPos(1, 0, -1)
        },
        {
            Side.BottomRight,
            new CubicPos(1, -1, 0)
        },
        {
            Side.Bottom,
            new CubicPos(0, -1, 1)
        },
        {
            Side.BottomLeft,
            new CubicPos(-1, 0, 1)
        }
    };

    public const float Sqrt3 = 1.7320508076f;

    private AxialPos canonical;

    public CubicPos Cube
    {
        get
        {
            return new CubicPos(
                canonical.X,
                -canonical.X - canonical.Y,
                canonical.Y
            );
        }
        set
        {
            canonical = new AxialPos(value.X, value.Z);
        }
    }

    public AxialPos Axial
    {
        get
        {
            return canonical;
        }
        set
        {
            canonical = value;
        }
    }

    public OffsetPos Offset
    {
        get
        {
            return new OffsetPos(
                canonical.X,
                canonical.Y + (canonical.X / 2)
            );
        }
        set
        {
            canonical = new AxialPos(
                value.X,
                value.Y - (value.X / 2)
            );
```

```
    }
}

public Hex (CubicPos fromCube) : this()
{
    this.Cube = fromCube;
}

public Hex (AxialPos fromAxial) : this()
{
    this.Axial = fromAxial;
}

public Hex (OffsetPos fromOffset) : this()
{
    this.Offset = fromOffset;
}

public bool Equals (Hex obj)
{
    return this == (Hex)obj;
}

public override bool Equals (object obj)
{
    return obj is Hex && this == (Hex)obj;
}

public override int GetHashCode ()
{
    return canonical.GetHashCode();
}

public static bool operator == (Hex lhs, Hex rhs)
{
    return lhs.canonical == rhs.canonical;
}

public static bool operator != (Hex lhs, Hex rhs)
{
    return !(lhs.canonical == rhs.canonical);
}

public static Hex operator + (Hex lhs, Hex rhs)
{
    return new Hex(lhs.Cube + rhs.Cube);
}

public static Hex operator - (Hex lhs, Hex rhs)
{
    return new Hex(lhs.Cube - rhs.Cube);
}

/// <summary>
/// Returns the clockwise next corner point of specified side in
/// 2D projected coordinates.
/// I.e., if Side is Side.TopRight, the returned corner will be on
/// the right-hand side of the flat-top hexagon.
/// </summary>
/// <returns>The corner position, in grid scaled with radius</returns>
/// <param name="side">The side for which to get corner point</param>
/// <param name="radius">Radius of hex cells in grid</param>
public Vector2 GetCorner (Side side, float radius)
{
```

```
    float yDelta = (radius * Sqrt3) * 0.5f;
    Vector2 center = GetCenter(radius);

    switch (side)
    {
        case Side.TopLeft:
            return new Vector2(
                center.x - radius * 0.5f,
                center.y - yDelta);
        case Side.Top:
            return new Vector2(
                center.x + radius * 0.5f,
                center.y - yDelta);
        case Side.TopRight:
            return new Vector2(center.x + radius, center.y);
        case Side.BottomRight:
            return new Vector2(
                center.x + radius * 0.5f,
                center.y + yDelta);
        case Side.Bottom:
            return new Vector2(
                center.x - radius * 0.5f,
                center.y + yDelta);
        case Side.BottomLeft:
            return new Vector2(center.x - radius, center.y);
    }
    return center;
}

/// <summary>
/// Returns the centerpoint of cell in 2D projected
/// coordinates, scaled with the radius value.
/// </summary>
/// <returns>The center point scaled with radius</returns>
/// <param name="radius">Radius of cells to scale
/// the point with</param>
public Vector2 GetCenter (float radius)
{
    float yDelta = (radius * Sqrt3) * 0.5f;
    return new Vector2(
        Axial.X * radius * 1.5f,
        (2 * Axial.Y + Axial.X) * yDelta);
}

public void SetFromScreenCoordinate (int x, int y, int size)
{
    float q = x * (2 / 3.0f) / (float)size;
    float r = (-x / 3 + Sqrt3 / 3.0f * y) / (float)size;

    float lx = q;
    float ly = -q - r;
    float lz = r;

    int rx = Mathf.RoundToInt(lx);
    int ry = Mathf.RoundToInt(ly);
    int rz = Mathf.RoundToInt(lz);

    float x_diff = Mathf.Abs(rx - lx);
    float y_diff = Mathf.Abs(ry - ly);
    float z_diff = Mathf.Abs(rz - lz);

    if (x_diff > y_diff && x_diff > z_diff)
        rx = -ry - rz;
    else if (y_diff > z_diff)
```

```
                ry = -rx - rz;
            else
                rz = -rx - ry;

            this.Cube = new CubicPos(rx, ry, rz);
        }

        public override string ToString ()
        {
            return string.Format (
                "[Hex: Cube={0}, Axial={1}, Offset={2}]",
                Cube, Axial, Offset);
        }
    }
}
```

AxialPos.cs

```
namespace Utilities.Hex
{
    /// <summary>
    /// Axial position container for hex coordinates
    /// </summary>
    public struct AxialPos
    {
        public int X { get; private set; }

        public int Y { get; private set; }

        public AxialPos (int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public override bool Equals (object obj)
        {
            return obj is AxialPos && this == (AxialPos)obj;
        }

        public override int GetHashCode ()
        {
            return (X.GetHashCode () << 8) ^ (Y.GetHashCode ());
        }

        public static bool operator == (AxialPos lhs, AxialPos rhs)
        {
            return lhs.X == rhs.X && lhs.Y == rhs.Y;
        }

        public static bool operator != (AxialPos lhs, AxialPos rhs)
        {
            return !(lhs == rhs);
        }

        public override string ToString ()
        {
            return string.Format ("[AxialPos: X={0}, Y={1}]", X, Y);
        }
    }
}
```

CubicPos.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

namespace Utilities.Hex
{
    /// <summary>
    /// Cubic position container for hex coordinates
    /// </summary>
    public struct CubicPos
    {
        public int X { get; private set; }

        public int Y { get; private set; }

        public int Z { get; private set; }

        public CubicPos (int x, int y, int z)
        {
            this.X = x;
            this.Y = y;
            this.Z = z;
        }

        public static CubicPos operator + (CubicPos lhs, CubicPos rhs)
        {
            return new CubicPos (lhs.X + rhs.X, lhs.Y + rhs.Y, lhs.Z + rhs.Z);
        }

        public static CubicPos operator - (CubicPos lhs, CubicPos rhs)
        {
            return new CubicPos (lhs.X - rhs.X, lhs.Y - rhs.Y, lhs.Z - rhs.Z);
        }

        public static CubicPos operator * (CubicPos lhs, int rhs)
        {
            return new CubicPos (lhs.X * rhs, lhs.Y * rhs, lhs.Z * rhs);
        }

        public override string ToString ()
        {
            return string.Format ("[CubicPos: X={0}, Y={1}, Z={2}]", X, Y, Z);
        }

        public float DistanceTo(CubicPos other)
        {
            return (Mathf.Abs(X - other.X) +
                Mathf.Abs(Y - other.Y) +
                Mathf.Abs(Z - other.Z)) / 2;
        }
    }
}
```

OffsetPos.cs

```
namespace Utilities.Hex
{
    /// <summary>
    /// Offset position container for hex coordinates
    /// </summary>
    public struct OffsetPos
    {
        public int X { get; private set; }

        public int Y { get; private set; }

        public OffsetPos (int x, int y)
        {
            this.X = x;
            this.Y = y;
        }

        public override bool Equals (object obj)
        {
            return obj is OffsetPos && this == (OffsetPos)obj;
        }

        public override int GetHashCode ()
        {
            return (X.GetHashCode () << 8) ^ (Y.GetHashCode ());
        }

        public static bool operator == (OffsetPos lhs, OffsetPos rhs)
        {
            return lhs.X == rhs.X && lhs.Y == rhs.Y;
        }

        public static bool operator != (OffsetPos lhs, OffsetPos rhs)
        {
            return !(lhs == rhs);
        }

        public override string ToString ()
        {
            return string.Format ("[OffsetPos: X={0}, Y={1}]", X, Y);
        }

        public bool InRange(float x1, float y1, float x2, float y2)
        {
            return (X >= x1) && (Y >= y1) && (X <= x2) && (Y <= y2);
        }
    }
}
```

Appendix 6: Source Code of Inference Engine Prototype

Note: The rule condition parsing uses a modified version of the B83 Parser (available at http://wiki.unity3d.com/index.php/ExpressionParser), which is not included in the listing. The modifications add support for equality, inequality, and the basic AND and OR operators of Boolean algebra to the formulas. Those modifications assume values of 0.0 for false and 1.0 for true in this context (Rule.Matches method uses threshold value of 0.5 to dictate whether the condition matches or not).

InferenceEngine.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;
using System;

/// <summary>
/// A simple inference engine prototype implemented in Unity.
/// Uses a Ruleset asset to define rules, and has editor
/// mode execution support.
/// </summary>
public class InferenceEngine : MonoBehaviour
{
    /// <summary>
    /// A single fact in the database implemented
    /// as key (identifier)-value pair.
    /// </summary>
    [Serializable]
    public class Fact
    {
        public string identifier;
        public string value;
    }

    /// <summary>
    /// The working memory implementation for inference engine
    /// </summary>
    [Serializable]
    public class WorkingMemory
    {
        /// <summary>
        /// List of all active facts in the working memory
        /// </summary>
        public List<Fact> facts;

        /// <summary>
        /// Reference to the expression parser instance
        /// </summary>
        public B83.ExpressionParser.ExpressionParser parser;

        /// <summary>
        /// Clear this working memory
        /// </summary>
        public void Clear()
```

```
{
    facts.Clear();
    parser = new B83.ExpressionParser.ExpressionParser();
}

/// <summary>
/// Apply one rule effect to the working memory.
/// </summary>
/// <param name="effect">Rule effect to apply.</param>
/// <returns><c>true</c>, if working memory was changed,
/// <c>false</c> if no changes were made.</returns>
public bool Apply(Ruleset.RuleEffect effect)
{
    switch (effect.type)
    {
        case Ruleset.RuleEffectType.SetIdentifier:
            if (TestSetFact(effect.identifier, effect.value))
                return true;
            break;
        case Ruleset.RuleEffectType.RemoveIdentifier:
            if (TestSetFact(effect.identifier, null))
                return true;
            break;
    }
    return false;
}

/// <summary>
/// Internal method to apply fact to working memory.
/// </summary>
/// <param name="identifier">Fact identifier.</param>
/// <param name="value">Value to apply. If this is null,
/// the fact will be removed from working memory</param>
/// <returns><c>true</c>, if working memory was changed,
/// <c>false</c> if no changes were made.</returns>
private bool TestSetFact(string identifier, string value)
{
    // Update expression parser
    if (value != null)
    {
        parser.AddConst(identifier, () => Double.Parse(value));
    }
    else
    {
        parser.RemoveConst(identifier);
    }

    // Update working memory
    var oldFact = facts.FirstOrDefault(f =>
        f.identifier == identifier);
    if (oldFact != null)
    {
        if (oldFact.value == value)
        {
            // The fact already exists with given value, no change
            return false;
        }
        else
        {
            // The fact exists but has different value, change it
            oldFact.value = value;
            return true;
        }
    }
```

```
        else
        {
            // Add new fact
            facts.Add(new Fact {
                identifier = identifier,
                value = value });
        }
        return true;
    }
}

public Arbiter arbiter;
public Ruleset activeRules;
public WorkingMemory workingMemory = new WorkingMemory();
public bool finished;

/// <summary>
/// List of rules that were matched during the most recent iteration
/// </summary>
private List<Ruleset.Rule> matchedRules = new List<Ruleset.Rule>();

public void Awake()
{
    Reset();
}

/// <summary>
/// Reset the inference engine. Clears the working memory.
/// </summary>
[ContextMenu ("Reset inference engine")]
public void Reset()
{
    workingMemory.Clear();
    finished = false;
    Debug.Log("Inference engine has been reset");
}

/// <summary>
/// Runs one iteration of the inference engine.
/// </summary>
[ContextMenu ("Run one inference engine iteration")]
public void RunIteration()
{
    if (finished)
    {
        Debug.LogError("Inference engine finished, reset to run again");
        return;
    }

    // Clear list of previously matched rules
    matchedRules.Clear();

    // Check which rules match the current working memory
    foreach (var rule in activeRules.rules)
    {
        if (rule.Matches(workingMemory))
            matchedRules.Add(rule);
    }

    if (matchedRules.Count > 0)
    {
        // Use arbiter to pick which of the matched rules will be fired
        Ruleset.Rule rule = arbiter.PickAndApplyRule(
```

```
                matchedRules, workingMemory);

            // If rule was returned, working memory was changed
            if (rule != null)
            {
                Debug.Log("Applied rule " + rule + " to working memory");
            }
            else
            {
                Debug.Log("No changes in working memory, finishing");
                finished = true;
            }
        }
        else
        {
            // No more matching rules, finish
            Debug.Log("No rules matched, finishing");
            finished = true;
        }
    }
}
```

Arbiter.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;

/// <summary>
/// A simple arbiter component of the inference engine
/// </summary>
public class Arbiter : MonoBehaviour
{
    /// <summary>
    /// Arbiter type used for rule matching
    /// </summary>
    public enum ArbiterType
    {
        FirstMatch,
        Random
    }

    public ArbiterType type;

    /// <summary>
    /// Superclass of the rule matching methods. Should be
    /// overridden to provide different rule-matching types.
    /// </summary>
    abstract class ArbiterMethod
    {
        protected InferenceEngine.WorkingMemory workingMemory;

        public ArbiterMethod(InferenceEngine.WorkingMemory workingMemory)
        {
            this.workingMemory = workingMemory;
        }

        public abstract Ruleset.Rule PickAndApplyRule(
            ICollection<Ruleset.Rule> rules);
    }
```

```
/// <summary>
/// First-match rule arbiter matching method. Iterates though
/// the given rules until one of them has effect on working memory.
/// </summary>
class ArbiterMethodFirstMatch : ArbiterMethod
{
    public ArbiterMethodFirstMatch(
        InferenceEngine.WorkingMemory workingMemory) :
        base(workingMemory) {}

    public override Ruleset.Rule PickAndApplyRule(
        ICollection<Ruleset.Rule> rules)
    {
        foreach (var rule in rules)
        {
            if (rule.ApplyTo(workingMemory))
                return rule;
        }
        return null;
    }
}

/// <summary>
/// Random-match rule arbiter mathod. Iterates the given rules in
/// random order until one of them has effect on working memory.
/// </summary>
class ArbiterMethodRandom : ArbiterMethod
{
    public ArbiterMethodRandom(
        InferenceEngine.WorkingMemory workingMemory) :
        base(workingMemory) {}

    public override Ruleset.Rule PickAndApplyRule(
        ICollection<Ruleset.Rule> rules)
    {
        // Randomize the rule collection using LINQ OrderBy and Random
        var rnd = new System.Random();
        var randomizedList = rules.ToList().OrderBy(i => rnd.Next());
        foreach (var rule in randomizedList)
        {
            if (rule.ApplyTo(workingMemory))
                return rule;
        }
        return null;
    }
}

/// <summary>
/// Creates an instance of the currently active arbiter
/// matching method type, and uses that to pick and fire
/// a rule from the rule list.
/// </summary>
/// <returns>The rule that was applied. May be null if
/// no changes were made to the working memory.</returns>
/// <param name="rules">List of available rules to apply.</param>
/// <param name="workingMemory">The working memory instance.</param>
public Ruleset.Rule PickAndApplyRule(ICollection<Ruleset.Rule> rules,
    InferenceEngine.WorkingMemory workingMemory)
{
    if (rules.Count == 0)
        return null;

    switch (type)
    {
```

```
            case ArbiterType.FirstMatch:
                return new ArbiterMethodFirstMatch(workingMemory).
                    PickAndApplyRule(rules);
            case ArbiterType.Random:
                return new ArbiterMethodRandom(workingMemory).
                    PickAndApplyRule(rules);
        }
        return null;
    }
}
```

Ruleset.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using System.Linq;

/// <summary>
/// The Ruleset ScriptableObject defines an unity asset, which is
/// used to configure rule sets for the inference engine.
/// </summary>
[CreateAssetMenu(fileName = "New Ruleset",
    menuName = "RuleSystem/Ruleset", order = 1)]
public class Ruleset : ScriptableObject
{
    /// <summary>
    /// Rule effect type, each effect can set or clear
    /// a given fact in the working memory.
    /// </summary>
    [Serializable]
    public enum RuleEffectType
    {
        SetIdentifier,
        RemoveIdentifier
    }

    /// <summary>
    /// Rule effect. Contains type, fact identifier and fact value.
    /// </summary>
    [Serializable]
    public class RuleEffect
    {
        public RuleEffectType type;
        public string identifier;
        public string value;

        public override string ToString ()
        {
            return "[" + type + "]" + identifier + "=" + value;
        }
    }

    /// <summary>
    /// One individual rule in the rule set. Contains list of
    /// conditions which must be satisfied for the rule to fire,
    /// and list of rule effects which will be applied to the
    /// working memory if the conditions were met.
    /// </summary>
    [Serializable]
```

```
public class Rule
{
    /// <summary>
    /// Conditions of the rule. Each condition is an expression
    /// that can be resolved into boolean value of either
    /// 0 or 1. For example:
    ///
    ///     a==1
    ///     b!=0
    ///
    /// Any previously set identifiers in working memory can
    /// be used in the expressions.
    /// </summary>
    public string[] conditions;

    /// <summary>
    /// Effects of the rule
    /// </summary>
    public RuleEffect[] effects;

    /// <summary>
    /// Matches the rule conditions against the working memory.
    /// </summary>
    /// <param name="memory">The working memory instance.</param>
    /// <returns><c>true</c>, if all conditions match the
    /// working memory, <c>false</c> otherwise.</returns>
    public bool Matches(InferenceEngine.WorkingMemory memory)
    {
        foreach (var condition in conditions)
        {
            if (memory.parser.Evaluate(condition) < 0.5f)
                return false;
        }
        return true;
    }

    /// <summary>
    /// Apply effects to the working memory
    /// </summary>
    /// <returns><c>true</c>, if the working memory was changed,
    /// <c>false</c> otherwise.</returns>
    /// <param name="memory">The working memory instance.</param>
    public bool ApplyTo(InferenceEngine.WorkingMemory memory)
    {
        bool changed = false;
        foreach (var effect in effects)
        {
            changed |= memory.Apply(effect);
        }
        return changed;
    }

    public override string ToString ()
    {
        return "[Rule: Conditions:(" +
            String.Join(",", conditions) + "), Effects:(" +
            String.Join(",", effects.Select(c => c.ToString()).
                ToArray()) + ")]";
    }
}

/// <summary>
/// Root rule container of the ruleset
/// </summary>
```

```
    public Rule[] rules;
}
```

InferenceEngineEditor.cs

```
using UnityEngine;
using UnityEditor;
using System.Collections;
using System.Collections.Generic;

/// <summary>
/// Custom editor for the inference engine. Reformats
/// the working memory to a more readable format.
/// </summary>
[CustomEditor(typeof(InferenceEngine))]
public class InferenceEngineEditor : Editor
{
    public override void OnInspectorGUI()
    {
        InferenceEngine myTarget = (InferenceEngine)target;
        EditorGUI.BeginChangeCheck ();

        // Basic Unity editor fields for arbiter and ruleset
        myTarget.arbiter = EditorGUILayout.ObjectField("Arbiter",
            myTarget.arbiter, typeof(Arbiter), true) as Arbiter;
        myTarget.activeRules = EditorGUILayout.ObjectField("Ruleset",
            myTarget.activeRules, typeof(Ruleset), true) as Ruleset;

        // Show finished flag as a label
        EditorGUILayout.LabelField("Finished",
            myTarget.finished ? "Yes" : "No");

        // Display the working memory.
        EditorGUILayout.LabelField("Working memory", EditorStyles.boldLabel);
        EditorGUI.indentLevel++;
        var facts = myTarget.workingMemory.facts;
        if (facts.Count == 0)
        {
            // Working memory empty
            EditorGUILayout.LabelField("(empty)");
        }
        else
        {
            // Show each fact as an indented label with identifier-value
            // pair as the label field label and value.
            foreach (var fact in facts)
            {
                EditorGUILayout.LabelField(fact.identifier, fact.value);
            }
        }
        EditorGUI.indentLevel--;

        // Save changes if needed
        if (EditorGUI.EndChangeCheck ())
        {
            serializedObject.ApplyModifiedProperties();
        }
    }
}
```

RuleEffectDrawer.cs

```
using UnityEditor;
using UnityEngine;

/// <summary>
/// Custom editor drawer for the rule effects.
/// Formats the effect field to a bit more
/// readable format.
/// </summary>
[CustomPropertyDrawer(typeof(Ruleset.RuleEffect))]
public class RuleEffectDrawer : PropertyDrawer
{
    public override void OnGUI(Rect position,
        SerializedProperty property, GUIContent label)
    {
        EditorGUI.BeginProperty(position, label, property);
        position = EditorGUI.PrefixLabel(position,
            GUIUtility.GetControlID(FocusType.Passive), label);

        var indent = EditorGUI.indentLevel;
        EditorGUI.indentLevel = 0;

        // Position the three properties horizontally
        var typeRect = new Rect(position.x, position.y, 100, position.height);
        var identifierRect = new Rect(
            position.x + 105, position.y, 50, position.height);
        var valueRect = new Rect(
            position.x + 160,
            position.y,
            position.width - 160,
            position.height);

        // Show editor fields for all three properties.
        var typeProperty = property.FindPropertyRelative("type");
        EditorGUI.PropertyField(typeRect, typeProperty, GUIContent.none);
        EditorGUI.PropertyField(
            identifierRect,
            property.FindPropertyRelative("identifier"),
            GUIContent.none);

        // If effect type is RemoveIdentifier, the "value" field is not needed
        // and can be hidden
        if (typeProperty.enumValueIndex !=
            (int)Ruleset.RuleEffectType.RemoveIdentifier)
        {
            EditorGUI.PropertyField(
                valueRect,
                property.FindPropertyRelative("value"),
                GUIContent.none);
        }

        EditorGUI.indentLevel = indent;
        EditorGUI.EndProperty();
    }
}
```