The path to the right decision: An investigation into using heuristic pathfinding algorithms for decision making in game AI

Ashley Smith

January 10, 2020

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras justo velit, vestibulum sit amet turpis in, interdum rhoncus magna. Proin pulvinar posuere iaculis. Duis vulputate tristique arcu, id pretium ante blandit ut. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Nam augue tellus, mattis quis consequat id, facilisis eu lectus. Vivamus euismod non quam sed condimentum. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Phasellus vitae consequat nisi. Morbi vulputate tellus ut nibh vulputate, vitae blandit ex faucibus.

Contents

Chapter 1

Introduction

1.1.1 Video games and artificial intelligence

Games are good for the global economy. Newzoo (2019) reports that in October 2019 the global games market was worth $148 billion. In order to remain competitive, developers are pushed to make bigger, better and more complex games. The evolution of the technologies available has allowed for a greater number of elements to be simulated in the game world which could potentially increase a game's worth (Blow, 2004). The entry barrier to creating these good-looking and well executed games has been lifted with the establishment of engines like Unity (Unity Technologies, 2005) and Unreal (Epic Games, 1998) to the point where even non-programmers can get involved using textual or visual scripting.

For many years, only the game's graphics were considered important (Blow, 2004; Yap, 2002), however, good physics systems or competent AI (Artificial Intelligence) are now recognised as a way of improving the end-user's experience just as much (Blow, 2004). Game AI is different from the normal, academic AI as it simulates behaviour and aims to be believable and fun, whereas an academic AI aims to achieve a level of intelligence or autonomy to excel at a given task (Nareyek, 2004, p. 60).

1.1.2 The need for game AI

AI has multiple uses within a games context but the majority of use cases employ AI to control the characters featured in a level of the game. Laird and VanLent (2001, p. 16) said that whether these characters are replacements for opposing players or characters that act as companions, villains and plot devices, "human-level AI can expand the types of

experiences people have playing computer games". Using AI opens up the opportunity for increasing the difficulty of the game, and could be perceived as the kind of challenges that make games fun (Buro, 2004, p. 2). Laird and VanLent (2001, p. 16) also hypothesised that utilising such an AI is a good step towards the development of enjoyable and challenging gameplay, and potentially to "completely new genres" (Laird & VanLent, 2001, p. 17).

"Customers value great AI" (Nareyek, 2004, p. 60) and so it's important to choose a suitable approach that fulfils the expectations of the player thus the requirements of the game (Millington, 2019, p. 19). Academic AI can be made using algorithms inspired by biology such as neural networks or genetic algorithms and trained through iteration or with datasets. These approaches aren't used in game AI because of the high requirements to train the AI to interact with a specific game (Nareyek, 2004, p. 64), moreover, it is easy to train the AI to play too strongly ruin the game (Tozour, 2002, p. 13). Instead, game AI developers embrace simpler, non-learning algorithms due to them being easier to understand, implement and debug (Tozour, 2002, p. 7).

1.1.3 Approaches to game AI development

The most basic forms of AI used in games take the form of a series of if-then statements and are known as 'production rule systems' (Tozour, 2002). These statements are organised in a list and the AI uses the behaviour of the rules that evaluate to *true*. The result is a very basic AI that is not only limited to what actions it can take but also when it can take them. Similarly, decision trees combine the same if-then style with branching structures to create game AI (Nareyek, 2004, p. 62), and the tree and subtrees are recursively traversed until a leaf node is found with the desired behaviour. This process is very easy to understand and implement (Millington, 2019, p. 295), and the branching structure makes visualisation of the process more intuitive than the basic list used in a production rule system. Because of this, many consider decision trees to be one of, if not the simplest techniques to making AI (Millington, 2019, p. 295; Tozour, 2002, p. 7).

TODO: DIAGRAM OF DECISION TREES HERE

FSMs, or Finite State Machines, are the most common approach to game AI (Orkin, 2006, p. 1; Millington, 2019, p. 309) due to being easy to understand and the efficacy of their output. FSMs consist of a directed graph where each node represents a state and the edges represent the transitions between them (Tozour, 2002, p. 6). A character can only be in one state at time and has no memory of any previous states (Colledanchise, Marzinotto,

& Ögren, 2014); each state represents an expected behaviour and determines what they do and the conditions to switch to a different state (Diller, Ferguson, Leung, Benyo, & Foley, 2004, p. 3). In the right environment, the impression of a well thought out FSM could compete with that a neural network, at a fraction of the time and resource costs, despite not always arriving at the optimal decisions (Sweetser & Wiles, 2002). However, each new behaviour requires the creation of a new state and the conditions of which this state integrates and transitions into other states, making expansion and maintenance cumbersome (Sweetser and Wiles, 2002, p. 2; Lim, Baumgarten, and Colton, 2010, p. 3). There's no easy way to combine the tests inside of FSMs and selecting the conditions for a state transition is still very much a process that must be done by hand (Millington, 2019, p. 313).

TODO: FSM DIAGRAM

The need for more flexibility in game AI has lead to the creation adaptation of modular algorithms such as behaviour trees (Lim et al., 2010, p. 1). Like decision trees, the recursive structure of a behaviour tree is simple to understand and implement while also being high level, allowing for more sophisticated AI to be created in a modular fashion through the use of subtrees and different node types (Shoulson, Garcia, Jones, Mead, & Badler, 2011, p. 144), each performs an action or check and then proceeds to succeed or fail (Lim et al., 2010, p. 4). It is these types that make the AI process at a higher level than standard decision trees, and when combined with leaf nodes that perform checks and actions to build trees, and then combined again to make trees containing subtrees, the simplicity and elegance of this technique certainly demonstrates why behaviour trees are getting attention (Shoulson et al., 2011, p. 144).

TODO: BEHAVIOUR TREE DIAGRAM

1.1.4 The relationship of pathfinding and game AI

One common requirement for game AI is for the characters to be able to traverse the areas of the game in a way which meets the player's expectations logically and efficiently — a task known as pathfinding. Regardless of what an AI decides to do, a pathfinding mechanic needs to be in place to allow the AI to navigate to where it needs to go, manoeuvring around obstacles while still taking a sensible route (Graham, McCabe, & Sheridan, 2003, p. 60). For most games, the algorithm of choice is A* as it is the de-facto standard pathfinding algorithm (Millington, 2019, p. 197; Botea, Müller, and Schaeffer, 2004, p. 2;

Nareyek, 2004, p. 64; Leigh, Louis, and Miles, 2007, p. 73).

The A* algorithm analyses the game's map and generates a path from one location to another while minimising a *cost* value — this value can represent anything but usually it represents the time or distance to travel along a given route (Yap, 2002, p. 44). This means that the pathfinding algorithm itself doesn't decide where to go, only how to get there and the manner in which it does so. When asked to calculate a path, a pathfinding algorithm is provided a graph of nodes to determine which nodes can be reached from which (Nareyek, 2004, p. 61). The algorithm isn't concerned in what the data represents (2D or 3D coordinate data), the data's form (a tree, a graph or a list of connections) or the unit of measurement to calculate the weight of an edge (length, traversal time or monetary cost), as long as it is equipped with the right functionality to digest this information (Millington, 2019, p. 277; Graham et al., 2003, p. 60).

With the pathfinding process taking place after the decision has been made, the opportunity to involve the data gathered from pathfinding algorithm is missed. Often, the AI will decide to approach the nearest object, but obstacles in the way mean that the cost of navigating to the destination is greater than some alternative. While implementing the algorithm isn't difficult, ensuring the AI generates a path to the correct destination is difficult to do well (Forbus, Mahoney, & Dill, 2002). Perhaps on the way to the destination, the character has to also navigate past traps or other hazards where it will need to decide whether to avoid or pass through — decisions that A* isn't fully equipped to deal with on it's own. If the AI wanted to factor in these hazards and real cost values, it would have to perform a more advanced check on where to travel too, maybe even using the pathfinding algorithm multiple times to definitely make sure that it wants to take the generated path, potentially ruining the game's speed and overall performance.

Pathfinding algorithms are actually general purpose search algorithms applied to a spatial context (Cui and Shi, 2011, p. 125; Orkin, 2003, p. 6; Yap, 2002, p. 46); there is no such restriction that these algorithms should be limited to pathfinding when in a games context. Millington (2019, p. 197) said that "pathfinding can also be placed in the driving seat, making decisions about where to move as well as how to get there". With search algorithms having the flexibility of being able to traverse graphs of nodes representing any kind of data, it's no stretch to imagine the A* search algorithm being applied to a graph containing the same tests and actions found in decision and behaviour trees in order to generate a 'path' of actions rather than a path of spatial data (Higgins, 2002, p. 114).

This is paper aims to investigate and re-engineer A* to make decisions rather than paths in a game context.

1.2 Literature Review

1.2.1 Dijkstra's algorithm: A graph and tree search algorithm

A search algorithm is a recursive method designed to find a match for a piece of data within a collection such as an array, graph or tree (Friedman, Bentley, & Finkel, 1976). A piece of data is provided and the search algorithm typically returns whether it is present and it's location. Dijkstra's algorithm (1959) is a search algorithm that operates on trees and graphs (which are then interpreted as trees). The algorithm calculates the shortest difference from any node on the graph to any other node, and can be terminated early to avoid unnecessary computation if a destination is provided and found.

Dijkstra's algorithm works through the recursive summation and comparison of distance values starting from the a given start node (Dijkstra, 1959, p. 269). Each neighbouring node is added to the *open* list, then, the current node's distance from the start is added to the length between the current node and its neighbour. If this tentative value is lower than the current distance value of the neighbour, it replaces it. When all the neighbours have been considered, the node with the lowest tentative value on the graph is selected and permanently 'visited' and are removed from the *open* list (Dijkstra, 1959). This process is repeated until either there are no *open* nodes left or the destination, if provided, has been visited, and thus a path and the distance from the start to the destination can be retrieved. The problem with Dijkstra's algorithm is that it always selects the node with the lowest tentative distance value, meaning that the algorithm has no notion of direction and is calculating the lowest distance to nodes that may not be relevant to getting to the destination (Millington, 2019, p. 214).

TODO: DIJKSTRA'S ALGORITHM DIAGRAM

1.2.2 A* algorithm: A heuristic best-first search algorithm

A* is an improvement of Dijkstra's algorithm in this regard (Hart, Nilsson, & Raphael, 1968, p. 101) — while it doesn't stray far from how Dijkstra's algorithm works in the sense that it operates using a tentative distance value and it keeps track of the nodes

that have and haven't been visited, it does extend the algorithm using what's known as a heuristic approach (Cui & Shi, 2011, p. 126). "Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal" (Pearl, 1984, p. 3). This means that in a pathfinding situation, a heuristic function could estimate the distance to the goal, by ignoring walls and measuring in a straight line, to direct the algorithm in the right direction and avoid evaluating routes that travel in the wrong direction to make the process more efficient (Cui & Shi, 2011, p. 127). Heuristics enable A* to perform a best-first search (Yap, 2002, p. 46), as the heuristic now has the power to select which node is the best to evaluate and prioritise over the others (Russell & Norvig, 2016, p. 94).

When identifying the next node to expand, A* will take this heuristic distance into account using the formula $f(n) = g(n) + h(n)$ (Hart et al., 1968, p. 102; Russell and Norvig, 2016, p. 95), where $g(n)$ is the real distance the algorithm has calculated from the start node to node n and is the distance to be minimised while finding the *goal*, $h(n)$ is the *heuristic* distance from the current node n and the destination, and $f(n)$ is the combination of these two metrics forming an estimate of the distance from the start node to the destination if travelling through node n, also known as the *fitness* value (Hart et al., 1968; Millington, 2019; Graham et al., 2003, p. 64).

This heuristic component of A* transforms it into a family of algorithms where applying a different heuristic selects a different algorithm (Hart et al., 1968, p. 107), moreover, implementing A* and using a heuristic that returns a constant value for all nodes reverts A* back into Dijkstra's algorithm (Lester, 2005, p. 10; Millington, 2019, p. 237). Conversely, implementing a well-designed heuristic method can be used to guarantee optimal solutions, and using a heuristic that is somewhere in-between can output results with varying degrees of accuracy in exchange for faster execution (Millington, 2019, p. 219). The implementation of a good heuristic can be difficult, as making the heuristic take more factors into account for accuracy has the drawback of making the algorithm less efficient overall with the heuristic being frequently used throughout the process.

On the other hand, Graham et al. (2003, p. 68) argue that one of the constraints of games the industry is the "over-reliance" on the A* pathfinding algorithm and describe the development of its many extensions as a way of avoiding the discovery of new techniques. The pathfinding process can require a lot of CPU resources, sometimes to the point of stalling the game, when applied to larger graphs (Cui and Shi, 2011, p. 127; Stentz, 1996,

p. 110; Graham et al., 2003, p. 67). This indicates that the performance of A* can vary depending on various factors, and is why it is important to optimise A* by selecting an suitable storage mechanism for the graph and internal node storage as well as using a reasonable heuristic that balances efficiency and efficacy (Millington, 2019, p. 228).

TODO: CRITICAL: A DIAGRAM*

1.2.3 Processing a character's perception of the game world

These approaches need a way to digest information about the character and its environment that is relevant to the decision making process, and selecting this information is just as important as the form it is delivered in (Cui & Shi, 2011, p. 126). This extraction of world data can vary in difficulty (Diller et al., 2004, p. 3) and is done for both performance and gameplay needs. All game AI solutions need the world to be re-interpreted to be better suited for both decision making (Buro, 2004, p. 2) and pathfinding (Diller et al., 2004, p. 3). Particular geometry of the map may need to be interpreted as vantage points, choke points or safe spots; particular formations of enemy units may need to be not only counted but also assessed for tactical strengths, whether engaging the units head-on is better than running away to a better location, and finally, the interpretation of time such as whether there is enough time to navigate to what would otherwise be a better location for fighting the enemy (Buro, 2004). An AI would then take this abstraction of the world, combine it with the character's data and then consider what the character should be doing — if the character is in an 'attacking' state and the player is nearby then the decision would involve getting into a position and hitting or shooting at the player.

If A* is to be used for game AI it also needs to operate on non-positional data. In standard pathfinding, each node is a position in the world and the edges between these nodes are the movements to get from one to another. Substituting each node to be an AI state and each edge to be an action that causes transitions from one state to another creates a graph that A* could process. The implementation the states and actions depend on the expectations of the AI — traditional pathfinding could be implemented by having an action that triggers movement and changes the position variable in the character's state. Regardless, a graph consisting of these nodes could be difficult to process and reduce the speed of A*.

A* can suffer from performance problems when used with many agents at once, or with a large or inefficient graph of nodes (Graham et al., 2003). Many solutions have been

discovered to prevent or reduce the impact on performance — one such solution is known as hierarchical pathfinding, where the game's map is simplified into chunks, (Cui & Shi, 2011, p. 126). By simplifying a group or grid of nodes into a single node that represents the whole group (like a quadtree), the pathfinding process can be applied to larger worlds as if they were actually smaller (Botea et al., 2004). This works well for standard pathfinding, but if A* is applied to something other than purely spatial world data then the nodes need extra consideration.

TODO: DIAGRAM ABOUT HIERARCHICAL PATHFINDING

There is a large amount of actions a character can take at any given moment (Nareyek, 2004, p. 62) and so applying a search algorithm to such a large set of tasks will make the process a lot slower than traditional methods if left unchecked. Hierarchical pathfinding essentially creates more nodes to expand in the short term to reduce the total number of nodes expanded in the long term and gain a net increase in performance. If hierarchical pathfinding reduces the amount of destinations, by collecting connections between the nodes in a given area, and simplifies navigation to and from this area then this technique can also be applied to the character's state so long as these connections can be aggregated using other metrics other than purely relying on their proximities to each other. One such way of grouping could be separating 'attacking' actions from 'defending' actions so that character's deciding to play defensively can eliminate one route of evaluation rather than multiple. Splitting gameplay elements like this can get complicated — if a certain unit in a strategy game can attack from a longer range, would attacking the enemy from a distance be a defensive manoeuvre or an offensive one, and how easy is this to change on a per-unit basis (Weber, Mateas, & Jhala, 2011)?

1.2.4 Making decisions with A* using GOAP

Orkin (2003, p. 11) expressed that expectations of AI are growing with the release of every new game and that "we need to look toward more structured, formalized solutions to creating scalable, maintainable and re-usable decision making systems". Techniques used to create the Game AI featured in most games do not contain the scalability Orkin envisions (Laird & VanLent, 2001, p. 17), however, Higgins (2002, p. 117) declares that a pathfinding engine can be created generically, especially if a templated language is used, to give the code even more reusability. Implementing A* in this way would allow it to be used for both regular pathfinding and anything else that could requires searching (Higgins,

2002, p. 120). With A* being efficient and optimisable (Millington, 2019, p. 215), reusing A* for game AI has the potential to bring the benefits it usually brings to pathfinding to AI while being adaptable enough to scale up and meet expectations.

Orkin (2006, p. 1) worked on the development of the game AI for the game F.E.A.R (Monolith Productions, 2005). The approach used for this AI was called GOAP which stands for 'Goal Oriented Action Planning' and uses the A* algorithm as part of its decision making process and "allows characters to decide not only what to do, but how to do it" (Orkin, 2003, p. 1). GOAP changes the data to be processed by A* from spatial data such as coordinates into character AI state data, such as what is found in FSMs. Therefore, the output of A* is no longer a sequence of movements but a sequence of actions also known as a plan (Orkin, 2003, p. 2; Tozour, 2002, p. 6).

GOAP essentially takes the idea of having state machines to encapsulate behaviours and replaces the hand-programming the conditions and connections of state transitions with a pathfinding process with the aim being the decoupling of states and their transitions (Orkin, 2003, p. 2). The pathfinding process uses each node to represent a state and the edges between each node as the actions that lead to those states (Orkin, 2003, p. 7) When GOAP is used, a goal is given for the character to achieve, which drives the AI to play the game rather than idly remain inactive, and what is returned is the sequence of actions that will satisfy the goal (Orkin, 2003, p. 1). An action is a representation of one thing the character will do to change the world in some way, like opening a door or picking up a weapon (Orkin, 2003); some actions have preconditions that require the execution of another action prior to it (Orkin, 2003, p. 5). A* will then find the sequence of actions, the plan, that satisfies the character's goal while minimising an arbitrary *cost* value — Orkin (2003, pp. 4–5) suggests that this process creates interesting character AI that can adapt to change while also having a code structure that is reusable, maintainable and "elegant".

A* being used in GOAP means that fundamental formula $f(n) = g(n) + h(n)$ (Hart et al., 1968, p. 102) is implemented in a way such that A* can perform a best-first search and expand the actions that are more likely to be useful to the situation (Orkin, 2003, p. 7). To calculate $h(n)$, Orkin (2003, p. 7) also shares that the heuristic in GOAP can be calculated as the summation of unmet conditions of the goal node, but this is rather unclear. Given the example in figure 1.1, "the goal is to kill an enemy" (Orkin, 2003, p. 3) could be interpreted in various ways that all seem unsuitable. If the goal condition is that

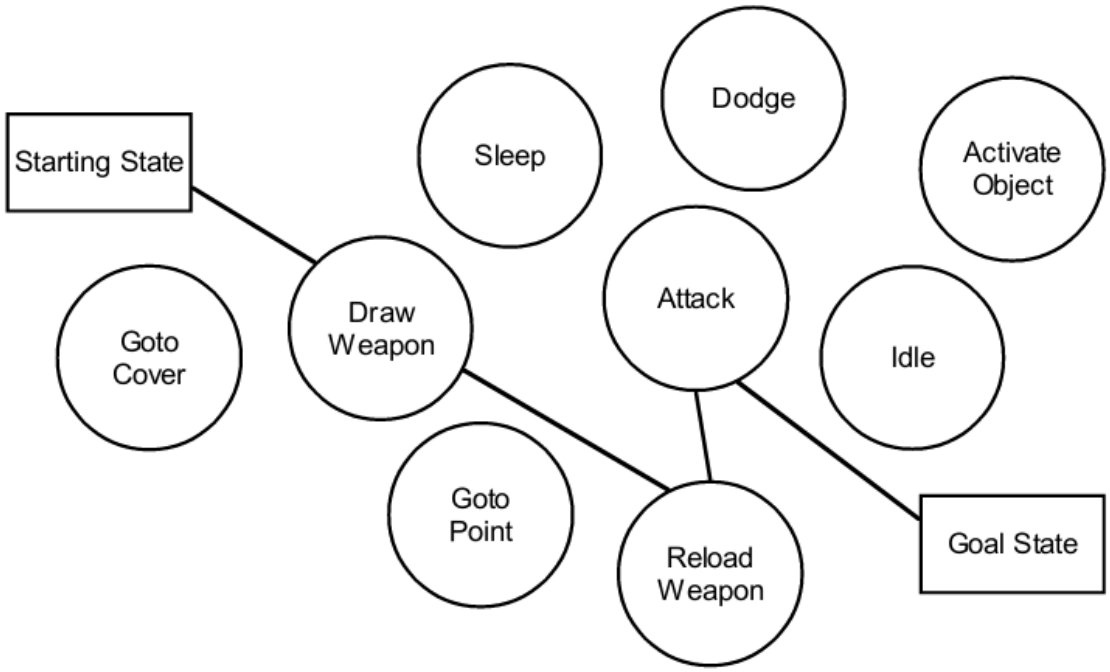

Figure 1.1: An example of GOAP (Orkin, 2003, p. 3)

the enemy is dead, the heuristic would be the same for all nodes but the goal state, making it potentially wasteful like Dijkstra's algorithm (Millington, 2019, p. 214). Alternatively, if the condition was divided into multiple conditions such as the weapon needing to be drawn, the weapon needing to be loaded, and the enemy being dead, it is implied that either the algorithm has been run previously to determine the requirements of killing an enemy, or that they were programmed by the programmer — how would the starting state know that the weapon needed to be reloaded before drawing it? While GOAP does seem elegant, the graph in figure 1.1 doesn't have a sizeable number of nodes; using Dijkstra's algorithm wouldn't incur much of a performance cost with a graph this size. Orkin (2003) suggests that a regressive search by starting at the goal and pathing to the start makes sense, and in figure 1.1, it does, as only one action can attack and so it's clear that the decision making process boils down to the character attacking the enemy.

Another problem of GOAP is that the high-level nature of the approach may not allow for enough control of the game (Stanciu & PETRUŞEL, 2012, p. 87) — while making abstractions of the world is necessary for AI to process the information (Buro, 2004, p. 2) and thinking on a higher level is beneficial for making more human-like decisions, the precision of the actions taken by the AI is both low level and very noticeable if incorrect (Graham et al., 2003, p. 60). Figure 1.1 shows a decision making process that decides what to do but not necessarily where to do it. When the action "Goto Point" in figure 1.1 gets

executed, there's no indication of how the destinations are generated other than satisfying the preconditions for another action (for example, moving to an object to activate it) (Orkin, 2003, p. 7).

This isn't specific to GOAP as it applies to every AI approach where the generation of these locations isn't part of the AI. Is the character's AI deciding where to go, or just deciding that it has to go somewhere? In order for "Goto Cover" to act in a similar way to the "Goto Point" action, a goal or action precondition would either have to require the character to be out of line-of-sight, or be using one of the designated cover spots — this is how F.E.A.R (2005) approached it (Orkin, 2006, p. 12). The former would require calculating the closest position that allows the escape of line-of-sight. The latter of these would require set locations to be marked as points of interest and then A* would be employed to find which cover spot would be the quickest to navigate to. Both of these would have their own problems, and wrapping the entire pathfinding process as a single action would mean that the information from the pathfinding request would not be used in the decision making at all. Without utilising the AI's ability to create an impression of thought, the information given to these actions could be poorly chosen and "may be perceived as a lack of intelligence by a human player" (Graham et al., 2003, p. 63).

1.2.5 Defining the notion of cost in the context of game AI

Cost, sometimes referred to as weight, is a term that will continue to be used when talking about A* as it is the the metric that governs the searching process (Graham et al., 2003, p. 60). *Cost* doesn't have to be a numeric value, as long as it can be compared and combined correctly with other *cost* values, however, one numeric restriction of cost is that it cannot, or rather should not, be negative. The method A* uses to determine if a route should be expanded before another is if its *cost* value is lower — when only positive values are added together it is assumed that there isn't a way for a *cost* to decrease in value. While in mathematics it is entirely possible and valid for these values to be negative, the problems that make this necessary are not applicable to games (Millington, 2019, p. 202).

In the formula $f(n) = g(n) + h(n)$ (Hart et al., 1968), each function returns a *cost* — the combination of the goal and heuristic *cost* values, also known as the fitness value, is used select the next open node to evaluate (Russell & Norvig, 2016, p. 94). If a and b are two nodes in a graph and $f(a) < f(b)$, A* will interpret this as node a being a more logical choice to expand (Orkin, 2003, p. 7). Node b won't be evaluated unless the route

derived from node a results in dead ends or more routes which have worse fitness values than node b. With the order of execution being so dependent on $f(n)$, it is important that the definition of the *cost* of an action is calculated correctly. It is difficult to design such a system when everything is arbitrary, but metrics such as distance can be used to prioritise actions with lower distances such as movement; resources such as gold or mana could be used in game specific interactions to minimise spending important resource, and the time spent doing the action can also be factored in to drive the character to resolve the situation as quickly as possible (Lester, 2005, p. 8). Millington (2019) said that "the cost function is a blend of many different concerns, and there can be different cost functions for different characters in a game" and that this is called "tactical pathfinding". The ability to change the *cost* per character further demonstrates the adaptability of A* — changing the way a character interacts with the world is just changing these *cost* functions so that the A* prioritises one action over another.

Some decisions are more troublesome to weigh than others. With the constraint of non-negativity, what would the *cost* be of an ability that regenerates mana instead of expending it? The only way apply reductions to values in this way would be to have a baseline *cost* for an action and then add or subtract from it, however, this does mean that this baseline value would dictate the maximum value of the reduction and so forward planning is necessary to ensure that all reductions can be applied in a balanced way. Another difficult type of decision to way are ones that don't have inherent characteristics; with a good goal for AI being unpredictable, surprising behaviour (Scott, 2002, p. 17), how would the incentive for performing strategies like flanking and ambushing be created, and how would it compare to the *cost* for attacking an enemy directly head-on? Orkin (2006, p. 14) said that in F.E.A.R (Monolith Productions, 2005) this behaviour just came naturally from using a dynamic decision making system such as GOAP and that while they were prepared to implement "complex squad behaviour", none was implemented — the squad's ability to ambush "emerged" from the combination of the "squad level" decisions and the individual character's decisions which gave the illusion of complex behaviour. Orkin doesn't mention much more on this subject and so GOAP's ability to surprise remains uncertain.

Harmon (2002, p. 405) suggests that an "opportunity cost" could be introduced which represents the other actions that cannot be taken as a result of taking the given, mutually exclusive action, the objective being to apply penalties to actions due to the fact that an alternative and possibly better course of action is available. While this makes sense when it is difficult to reward the AI for performing these strategies, but adding penalties

to other actions could get complex, there is no discussion of this concept from Orkin (2006) from his work on F.E.A.R. Although, it is unclear whether Harmon means to apply penalties based on the current action or the other actions, and if it's the latter, is this a flat *cost* shared amongst all actions or does it try to give smaller penalties to the better tasks somehow? Regardless, to calculate this *cost* some mechanism of the AI, such as the weighting function used to determine $g(n)$ or possibly the nodes themselves, needs to have knowledge of other nodes so that this opportunity cost can be determined and used, yet again requiring forward planning.

GOAP uses a simple "action cost" system to prioritise which tasks to carry out first (Orkin, 2006, p. 11). Using an integer or floating point value is ideal for the type of *cost* as it can be added and compared with other values trivially. However, GOAP's method of creating unique characters is to assign different actions to make them act differently (Orkin, 2006, p. 8) as opposed to having them evaluate situations differently as Millington (2019) suggested. Giving characters different actions will implicitly change how they prioritise the same situation as they will be given new routes to explore, and for most games this might be the best idea. It could be a good idea to split the *cost* type into a set of values to give actions different values. Throwing a grenade and shooting a pistol could be considered equivalent in difficulty to perform, but typically grenades have more firepower in exchange for there being less available; reducing these actions to a single value could be difficult depending on the approach, but splitting the value into a set would allow the programmers to keep track of firepower and usage penalties separately. When needed for comparison, this set would evaluate to the sum of it's parts — but certain characters or situations could choose to adjust or completely omit parts of the *cost* so that actions are perceived in different ways depending on how a character views a certain cost. For a fast character, the *cost* to move might be halved and the *cost* of being exposed doubled in order to give the impression of an agile and hard-to-hit character.

1.2.6 Generation, selection and application of goal state nodes

When the game world has been processed, the actions a character can perform have been laid out and the methods of evaluating courses of action have been provided, the only things remaining that A* needs to function are the start and goal node states for the actual decision. The starting state is trivial as it is simply the current state of the character and world; the goal state node requires more consideration than that though. In

standard pathfinding, the output is the shortest path from the starting node to the goal node, if such a path exists (Nareyek, 2004, p. 61). For game AI though, the goal needs to represent how the character or world should be — or rather the objective outcome of the decision making process. This objective can be difficult to ascertain as a goal node could represent anything; what seems to be a simple goal such 'win the game' becomes a rigorous series of tests to both calculate the *cost* reaching the goal than another (Harmon, 2002, p. 403). On the other hand, objectives that are too small or disconnected may not combine correctly to form this over-arching goal of winning the game. A balance is needed, whether that means the objective is to chase the player or defend an area, the objective needs to be focused on winning without being vague.

While Orkin (2003) talks about in great depth about how GOAP handles making a decision to satisfy goal conditions, but doesn't describe where these objectives come from. A "squad coordinator" is mentioned (Orkin, 2003, p. 13) that organises multiple agents into squads when they're close together, but the actual goals of the AI can come from both the character's individual AI and the squad coordinator. GOAP doesn't replace an FSM, so it could be inferred that the character's state when a goal is reached determines the next goal and thus the character's behaviour, with the initial state being defaulted, scripted or randomised (Orkin, 2003, p. 2). Alternatively, game AI could be created in layers, and the output of one layer could be the desired goal of another, but this would mean that this problem propagates to the highest level.

Another talking point regarding goals is the amount of designated goal state nodes in the graph. There is typically only a single goal in standard pathfinding, but it is possible, maybe even advantageous, for some games to contain multiple goal nodes in a graph (Millington, 2019, p. 272). Instead of checking for a match with a certain location (or state, when developing AI), a method that checks whether a given node meets the requirements of being classified as a goal could be used instead; the heuristic would also need revision as it would need to calculate the heuristic *cost* to reach the 'nearest' goal rather than just the single, given goal (Millington, 2019, p. 272). Having multiple goals and goal types (Higgins, 2002, p. 121) would grant the ability for the AI to re-route to a different goal if it's easier and therefore accomplish the same task in multiple different ways without creating generic goals; this has its drawbacks though, one being the need for a more intricate and potentially confusing implementation and design of the AI needs, the other being the creation of balancing difficulties to ensure goals are prioritised as expected.

1.2.7 Summary

Search algorithms such as A* are generic, maintainable and versatile and are therefore theoretically suitable replacements for FSMs and behaviour trees for implementing game AI. While GOAP does use A* for part of it's decision making process, it isn't a complete solution and still separates decision making from the pathfinding process. This is acceptable and valid as GOAP is for generating a sequence of actions whereas pathfinding is strictly for navigating the map in order to perform these actions. Unfortunately, some information found during pathfinding that could be considered useful for decision making is lost unless explicitly communicated — a decision might request to navigate to some location, but the path generated might be longer than expected and a different course of action could have been more appropriate. Without replanning, GOAP's disconnect between these systems could result in the wrong decisions being made.

In this paper, the mechanisms of the A* search algorithm are examined and re-engineered, through the substitution of input and output types, to investigate the modularity and adaptability of an AI that uses search algorithms to make decisions while actively involving pathfinding in the process as opposed to keeping these systems separate. Several approaches to defining goals and heuristic methods will be used to visualise the effects they have on a squad-controlling game AI. The aim of using this approach is to bring decision-making and pathfinding closer together and therefore simplifying the overarching process of perceiving, deciding and interacting in the game world.

1.3 Methods and Methodologies

1.3.1 Algorithm selection

As discussed, this paper will be using A* as the pathfinding and decision making algorithm of choice. A* is already used in many games for pathfinding mechanics (Millington, 2019, p. 197) and so it could be inferred that any developers intending to implement game AI using a search algorithm would prefer to reuse the A* algorithm specifically. A* is more of a family of algorithms rather than just one (Hart et al., 1968, p. 107) — implementing A* naturally grants access to using Dijkstra's algorithm (1959) when given a constant heuristic (Lester, 2005, p. 10). Similarly, providing different heuristic functions allows the adjustment of the effectiveness and efficiency of A*'s output (Hart et al., 1968, p. 107).

There are multiple adaptations and extensions of A* because of its wide usage. IDA* (Iterative Deepening A*) is an algorithm based on A* that eliminates the need for the internal storage (Korf, 1985, p. 36) to reduce the amount of memory used during execution at the potential cost of computational speed (Botea et al., 2004, p. 2; Yap, 2002, p. 44). This is done by creating a threshold initially set to $threshold = h(start)$ and terminating the current iteration when $f(n) > threshhold$, which then updates the next iteration's threshold to the minimum $f(n)$ that exceeded the the current threshold (Korf, 1985, p. 103). It is this repetition of expansion that can make this algorithm inefficient (Yap, 2002, p. 46), but the lack of overhead from storing nodes could have the opposite effect and make the algorithm run faster than A* in some situations (Korf, 1985, p. 106).

D* (Dynamic A*) is an adaptation of A* that operates under the assumption that the cost of navigating from one node to another can change, such as when the environment is only partially known (Stentz, 1997, pp. 1–2). The navigation of these environments require frequent map updates and the constant regeneration of paths, a process which is already expensive. D* aims to generate optimal paths efficiently for full, partial or no knowledge of the map (Stentz, 1997, p. 2). To do this, D* mainly features the functions $PROCESS-STATE$ and $MODIFY-COST$, the former to calculate paths to the goal and the latter to modify and register costs and their affected states (Stentz, 1997, p. 3).

TODO: MAYBE DIAGRAM COMPARING ALGORITHMS

These adaptations could be used instead of A*, however every extension of A* comes with its own benefits and drawbacks. Only the game AI developer knows which search algorithm is most relevant for their use case, therefore for the purposes of this investigation only the standard A* algorithm implementation will be used as it is the most commonly used algorithm and will be applicable to most games.

1.3.2 Language selection

For a lot of developers, the choice of programming language comes from the requirements of their chosen game engine. Unity (Unity Technologies, 2005) uses C# or JavaScript, Unreal (Epic Games, 1998) uses C++, and some engines implement their own language, one example being Godot (Linietsky & Manzur, 2014). Godot gives developers a choice between C++, C#, languages that have bindings written to Godot's GDNative module, and finally their own language called GDScript which writes like Python. Most engines take advantage of C++'s performance even if they don't use it as a scripting language —

Unreal, Unity and Godot all use C++ as part of their engine.

Unity (Unity Technologies, 2005) and Unreal (Epic Games, 1998) are both popular, so naturally C# and C++ are both used a lot in the industry. Blow (2004, p. 30) said that C++ was used in most games, most likely due to C++'s speed of code execution being useful for resource intensive engines and video games. C++ also features a template system which grants access to generic programming; templated classes and functions are only defined once but can be reused with different parameter types. This is very suitable for this paper's use case, while a degree of genericness can be achieved through base classes and virtual functions, C++ templates are an alternative that doesn't require the use of inheritance or the grouping of classes (Higgins, 2002, p. 117). Templated code is used to generate a new class or function for a given type at compile time, meaning that one implementation of the A* algorithm can be given parameters of any type have multiple uses throughout a single program (Higgins, 2002, p. 120).

A* could also be implemented with C#, but C#'s generics do not grant the same freedoms that C++'s templates do. C# is a higher level language, so elements such as memory management are either hidden or contained in abstractions in an effort to make these things simpler. The abstractions of high level languages can be reasonable and advantageous for some developers and games, but with pathfinding being a vital element to a lot of games and is therefore commonly built into game engines it could be said that features of a low level language like memory management or C++ templates are more applicable for implementing pathfinding algorithms. It is for this reason that C++ is to be used for the purposes of this research.

1.3.3 Tool and framework selections

To research game AI, a game environment needs to be created so that the AI's interactions can be observed. This requires making some sort of video game and so an appropriate tool needs to be chosen. Established game engines are viable candidates as they trivialise the process of programming the necessities for a standard video game: opening a window, loading resources, rendering the environment and handling input. Even though a simple text adventure can be programmed with very basic programming knowledge, creating and visualising the choices for the AI to make would be time consuming and hard to debug. A library like the Open Graphics Library, or OpenGL (Silicon Graphics, 1992), allows 2D and 3D graphics to be used in a video game and would make visualisation of the AI and

environment easier. Alternatively, using an established engine introduces technology such as debugging along with the basic functionality — this technology could be considered redundant use in research as simply running the project would require the installation of a relatively large piece of software.

SDL (Simple DirectMedia Layer, Lantinga, 1998) and SFML (Simple Fast Multimedia Library, Gomilia, 2007) are two libraries that could be considered a good compromise between low level graphics API usage and full-featured commercial engines. Both SDL and SFML implement essentials like opening a window, rendering graphics and responding to input without introducing unnecessary functionality. Programming a full game using these frameworks requires greater technical knowledge and effort than using a traditional game engine, but for the purposes of this research, fewer game systems are necessary and so using SDL and SFML is more acceptable. SDL is used as a starting point for game engines just as much as it is used for standalone games; when paired with a graphics API like OpenGL, SDL facilitates the creation of the game's window and input handling while providing basic rendering functionality with the expectation that the developer will use the graphics API to draw their game.

SFML on the other hand is more of a wrapper for OpenGL specifically, and contains an assortment of useful predefined classes to be used when rendering 2D graphics game. SFML is much larger when compared to SDL because of this, but for a basic game it would be nonsensical to reimplement low level graphical components just to get objects appearing in the game's window. With all things considered, SFML seems to be an intelligent choice for this experiment — it provides a suitable amount of basic and additional functionality for building a simple 2D game without requiring the installation of a game engine.

Finally, one additional feature this research will need is the tooling to fully create and observe the testing environment. Dear ImGui (Immediate mode GUI, Cornut, 2014) is a library that makes interfaces easy to implement; a level editor, entity inspector and debugging tool will help observe the AI as well as design the situations it will be placed in. Nuklear (Mettke, 2015) is an alternative to Imgui, but it focuses more on creating high quality interfaces for use in both games and tools, and so it can be customised in a greatly in both appearance and functionality. Since ImGui concentrates on being quick and simple and isn't intended to be used to create a polished interface, it is framework of choice for interfaces for this research.

1.3.4 Research methodologies

Millington (2019, p. 236) said that the only way to compare the effects of heuristic functions is to visualise the algorithm, and so the nature of this research is going to contain qualitative data from observation and inspection. There might be some quantitative aspects to this research, such as comparing the amount of wins or the amount of turns taken to win for each approach, but such values on their own have no meaning in the context of a video game when the level of difficulty, intelligence and randomness can directly influence them. These values may be useful for determining overall efficacy, but the majority of the data is going to need to be qualitative so that these values have context. This research seeks to answer to the following questions:

- How plausible is it to use A* in a decision making process for game AI?
- How does the inclusion of pathfinding affect the AI's decision making capabilities?
- How does changing the components used in A*'s fundamental formula affect the output of the algorithm?
- How easy is it to externally influence the AI's decisions, and how can aspects like difficulty be injected into the AI's decision making process?

These questions require careful study of the game AI and its actions, and so based on these questions Baxter and Jack (2008, p. 545) suggest that performing case studies will be the most effective research method for this investigation. In order to perform a successful case study, Baxter and Jack (2008, p. 546) also suggest to "bind" the case by explicitly stating what will and won't be studied in the case. This research is about the capabilities and the behaviour of a game AI created with A*, and so following information will *not* be studied in order to keep the study in the same scope:

- **The execution speed of each variation of the A* algorithm:** Unless significantly slow to the point where the game AI is rendered impractical, an accurate recording of the time it takes for the algorithm to finish processing will not be useful in evaluating the suitability of using A* for decision making and will not be recorded.
- **The number of games won by each game AI:** Each AI will not be necessarily playing the same number of matches, playing against the same opponents or playing

on the same maps — it is up to the researcher's discretion to determine which combinations will provide the most useful information. An estimation of the general ratio of wins and losses for an AI may be useful for describing and evaluating behaviour, but an accurate record of wins and losses is unnecessary for determining the overall fitness for use in a video game.

There are multiple types of case studies. This research will be studying the resulting game AI produced by multiple approaches to identify differences and similarities between them, which is the type of case study Yin (2003) describes as a "multiple-case" study, or alternatively known as a "collective" case study (Stake, 1995). Other types of case studies Yin (2003) mentions, such as explanatory and descriptive case studies, involve just a single case study that typically involves real life which isn't applicable to this study. Moreover, while you can include embedded units within a single case study to consider things outside of the selected case (Baxter & Jack, 2008, p. 550), it is more beneficial for this investigation to consider multiple cases in just as much detail and then consider them collectively (Baxter & Jack, 2008, p. 555). This means that a more reliable and robust understanding can be gathered of the overall case (Baxter & Jack, 2008, p. 550).

Each case in this collective study is going to involve altering part of the AI's programming and placing the resulting AI into various scenarios and seeing how it performs. Changing the heuristic function to prefer certain actions or changing the AI's interpretation of a goal are examples of such alterations, whereas the scenarios come in the form of both symmetrical and asymmetrical in-game maps featuring distinct unit and obstacle layouts. Each AI will then be observed in play against itself and other AIs on numerous maps, where each map can be modified for further exploration on a case by case basis. Descriptions of notable events and significant benefits, drawbacks and details will be recorded. Conducting the studies in this way will give an insight to the diversity of the different behaviours that can be achieved of making an AI this way, while also validating how functional each AI is in the context of a video game.

1.3.5 Software development methodology

To build the testing environment necessary for testing this experimental approach to making game AI, a software methodology needs to be adopted to keep things on track. There are multiple methodologies to choose from, each catering to different situations and having their own benefits and drawbacks:

Waterfall: Teams adopting the traditional waterfall method follow a rigid structure from planning to operations. Each step needs to be fully completed before moving on — Royce (1987, p. 330) shows that a waterfall process is flawed when revisiting previous steps is required to finish the process. The waterfall methodology is easy to understand, but lacks the flexibility of other methodologies.

Rapid Application Development: *TODO: GET BOOK FROM LIBRARY AND CITE THIS* (Martin, 1991). Mention how it aims to be lightweight and improve on waterfall.

DevOps: The DevOps methodology focuses on the deployment of software, not the development. DevOps encourages short and continuous releases for software with each release being planned, programmed and built based on the feedback from the previous (Bass, Weber, & Zhu, 2015). This research isn't releasing anything to an external party and therefore does not benefit from the DevOps methodology, however, one thing to note is that "DevOps emphasizes the relationship of DevOps practices to agile practices" (Bass et al., 2015, p. 15), meaning that it is common for teams using DevOps to also use agile methods during development.

Agile: According to The Agile Manifesto (Fowler, Highsmith, et al., 2001, p. 2), agile methods embrace responding to change as opposed to how waterfall's planning process is there to avoid changes. Agile also places emphasis on people, whether they are developers or clients, collaborating on a regular basis to review and deliver parts of a project during development (Fowler, Highsmith, et al., 2001, p. 3). There are frameworks like SCRUM (Fowler, Highsmith, et al., 2001, p. 1) that embody these same values, but being agile is about the team's recognition of these values and commitment to following them — any approach can be agile as long as these values are remembered.

This investigation requires the development of both an experimental AI and a game to observe its behaviour in, therefore it is hard to be certain about whether an approach will produce the right results, so a waterfall approach will not be used. The requirements and other elements of the testing environment may also change as new realisations are made, and so with this in mind, an agile approach will be used so that any unforeseen obstacles can be dealt with.

Chapter 2

Design, Development and Evaluation

2.1 Design

2.1.1 Requirements

In order to investigate the effects of combining the pathfinding and decision making processes, a set of requirements need to be satisfied by the testing environment. These requirements are designed to facilitate: the development and usage of a basic 2D application, the creation of the game AI variations and the scenarios they will be placed into, and finally the observation of the AI's behaviour in these scenarios. A testing environment would be considered adequate if it contains the following:

- **A program that renders 2D graphics and handles mouse clicks:** This is a fundamental requirement of any video game as stated in section 1.3.3. Making a game in 2D as opposed to making it text-based allows for a better variety of genres and mechanics and makes the game more visual. 3D graphics would allow for an even greater variety of gameplay, but aren't necessary for simple games like the one featured in this investigation.
- **A turn-based game with strategic depth:** A game AI needs an environment to interact with, which means building some sort of game. Sections 1.1.1 and 1.1.2 describe how game AI is used as an in-game device for offering the player more choices for interaction, implementing difficulty and making the game more enjoyable. There

isn't a restriction on the game needing to be turn-based, but it makes it easier to implement the ability to rewind the AI's turn easier, therefore making it easier to observe the AI's behaviour. Turn-based gameplay also acts as a limit for the AI; in a real-time game, pathfinding algorithms are typically run repeatedly so that the AI can regenerate the path to account for changes in the environment. In this simple turn-based game, no external factors will change the map or the status of any units during the AI's turn and the algorithm only need to generate a single plan.

The game needs to contain a variety of possible choices large enough to test both a player and an AI. Strategy games can easily give simple mechanics depth. The only real requirement in terms of gameplay is that players need to be able to perform more than one action per turn — *TODO: write about tic-tac-toe making this requirement?*

- **A map editor and a collection of maps:** Each case study is going to involve observing an AI's behaviour in an assortment of scenarios on different maps as stated in section 1.3.4. Therefore, a method of creating, editing, importing and exporting maps is important to ensure that each AI has enough opportunities for it's notable details to be displayed. Symmetrical maps showcase how having the first turn of a game changes the AI's gameplay, while asymmetrical maps reveal how well an AI takes advantage of one-sided maps. It is uncertain as to whether the AI will show interesting behaviour on every map, so the ability to edit a map while a game is in-progress will allow for greater inspection of any signs of behaviour of significance.

- **Implementation of a team controller:** A game usually becomes strategic when control is given over a number of units as opposed to a single character. In this context, the definition of a team controller would be the designated of owner of a collection of units — thus it is actually the team controllers that are playing against each other and not the units. The controller of a team should be replaceable so that other controllers can be observed for changes in behaviour. This is more of a dependency than a requirement; implementing a base team controller class with common functionality would simplify the interactions between AIs and the units they control. Further more, allowing one implementation of game AI to compete against another creates additional opportunities to observe the similarities and differences between them, which is partially the focus of the third and fourth research questions in section 1.3.4. Another bonus to using a base class for gameplay is that all controllers have to adhere to the same rules and information to keep the experiment fair.

- **Ability for a human to play:** While the focus of this research is for an AI to play, there are several benefits to allowing human interaction that justifies this requirement. Firstly, humans will be able to test the game for bugs and fundamental flaws while the game mechanics are being developed. Secondly, a match where a human can play against the AI would allow the human to test basic strategies to gauge the difficulty of the AI. Finally, giving a human the ability to assume control of a team while still in play would allow a researcher to probe the AI for different reactions in very specific moments during a scenario. Allowing a human player to choose which actions are invoked actions as opposed to having an algorithm generate a sequence of them would be as simple as overriding some of the team controller functionality specified in the previous requirement.

- **Implementation of trivial team controllers:** This requirement is about providing more ways to test the AI. It would be time-consuming to have a human play full-length matches against every variation of an AI, but on the other hand it would be difficult to learn from watching two AIs compete without some sort of base for comparison. To satisfy this requirement, two new team controllers should be added: one that simply ends their turn without doing anything, and another that selects moves randomly. The first team controller would be useful in testing an AI's intelligence and difficulty by having one team do nothing consistently, creating a way to compare the AI variations in a quantitative way, such as by counting the amount of actions required for the AI to win. The second team controller acts as a trivial opponent — by doing things randomly, this controller will mostly waste its turns while having the potential to perform actions that would be disadvantageous to the AI. This controller will produce demonstratory previews of the AI's overall behaviour, giving an insight into the AI's responses to random samples of gameplay.

- **A generic implementation of the A* algorithm:** A* is critical to the research questions listed in section 1.3.4 and is also a common core component in most games. Generically implementing A* means using technologies such as C++ templates to write multi-purpose code — each component of A* discussed in section 1.2.2 needs to be equipped to handle multiple types used for different purposes. This implementation of A* needs to use templates for the types of the *states*, *actions* and *costs*, while also accepting the following functions as arguments: the function for receiving nodes (*states*) connected to a given node; the function that checks whether a node is considered a goal; the heuristic function for finding $h(n)$; the edge weighing function

used to find $g(n)$; the function for applying an *action* to a *state*; and finally a *cost* comparison function used to compare $f(n)$, the fitness of a given node. This would allow A* to be used for not only standard pathfinding but also in the game AI's decision making process.

- **Implementation team controllers for each variation of AI made with the A* algorithm:** This requirement is important to both this investigation and for answering any of the research questions in section 1.3.4. Each team controller is expected to use the the A* algorithm in a different way so that when introduced to a scenario they can be played against and studied. Instancing a controller for each variation allows for a direct comparison between AIs in a single round. It is important to remember that strength is not equivalent to suitability and that an AI consistently winning does not prove anything. However, comparing the range of strengths between possible variations of AI will be used to draw conclusions with regards to the third and fourth research questions in particular. The most important observations to be made in AI versus AI rounds are the differences in tactics deployed and whether these tactics can be traced back to their implementation of A*.

- **A method of stepping through and rewinding an in-game turn:** Each game AI is going to be competing against other AIs in addition to the trivial controllers discussed above, meaning that unless a researcher sets up a human versus AI round, a round will be entirely autonomous with the possibility of instantaneously concluding. If the result of a round is surprising or if the AI needs to be studied further, it is vital that the history of events that took place can be explored and revisited. When combined with the requirements listed above, the researcher should also be able to intervene and switch a team controller into a human controller and resume play from any moment in time and see how the outcome of the round changes. Browsing the history of actions will be the most effective way of observing every part of an AI's turn and will be invaluable for understanding the AI's decision making process.

2.1.2 Designing the program's basic structure

The first requirement listed in section 2.1.1 is to create an interactive window that supports 2D graphics. Figure 2.1 illustrates an overview of the program; the *App* class manages the window and handles the main loop of processing input and rendering the screen. The *Scene* class can be inherited to introduce different functionality to the testing environment and

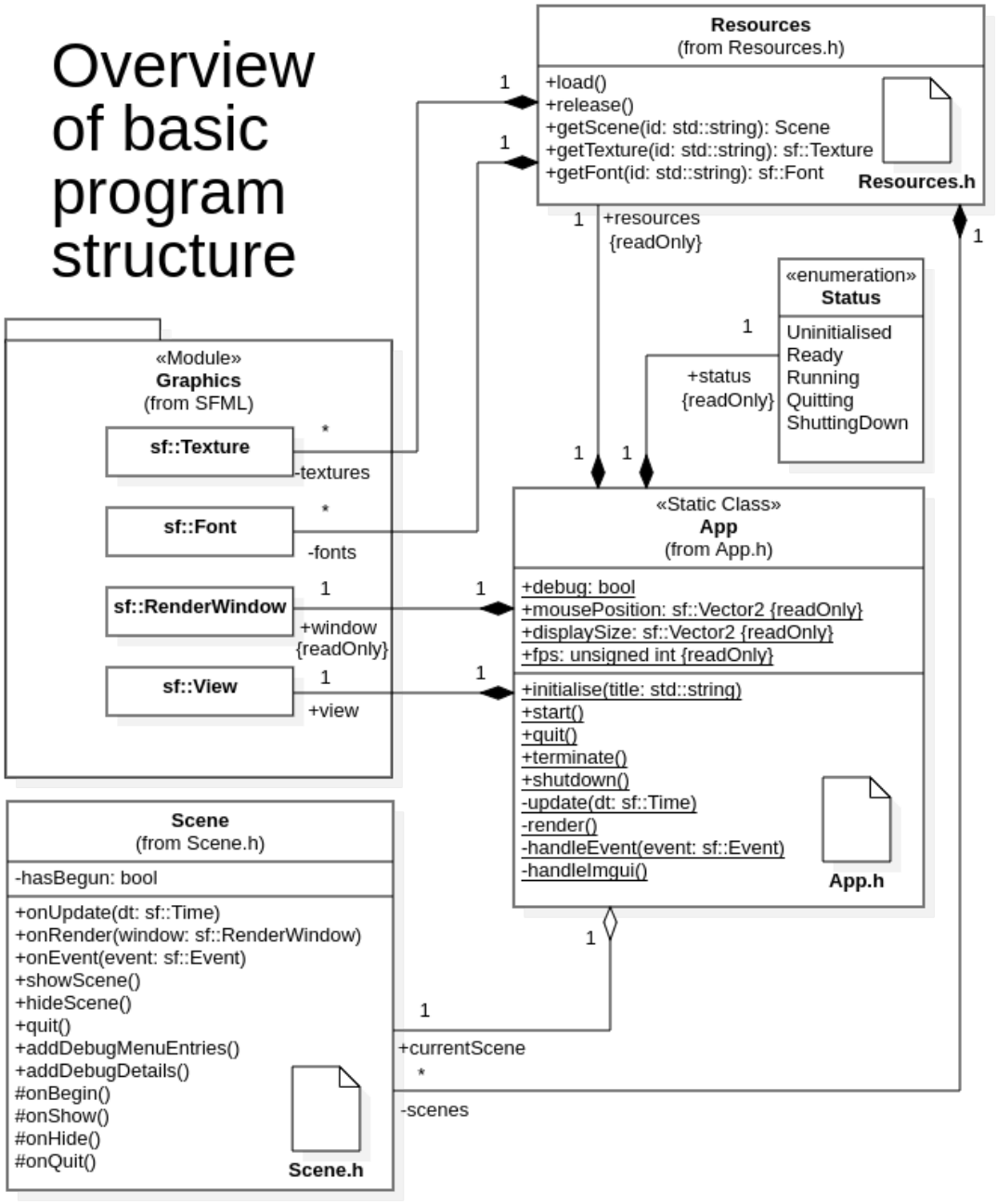

Figure 2.1: A UML diagram detailing the infrastructure of the testing environment

will be used to encompass the game and its logic; since *App* is entirely static, the game will have easy access to any of the game's variables or any data that has been loaded. *Resources* acts as a manager and will store fonts, textures, scenes and eventually game-related data like maps. The enumeration *Status* is used to determine what state the game is in — the app is initialised with a title with *initialise*, where it can then be given the first *Scene* to use for when the game enters the main loop with *start*. The *start* function will then run continuously until the program is closed or until a *Scene* calls *quit* function on game,

starting the process of releasing resources and performing any *Scene*-related functionality before terminating the main loop closing the application.

2.1.3 Designing the strategy game

The second requirement in section 2.1.1 is to build a simple, turn-based strategy game for the AI to play. The rules of this game are simple: each player controls a squad of units with the aim being to eliminate all enemy units while maintaining at least one surviving unit. During their turn, a player can select one of their units and then move and attack with it. A player has a set amount of MP and AP (movement and action points) per turn, and can distribute them between each of their units. Each unit drains a different amount of MP per tile when moving and a different amount of AP when attacking as shown in table 2.1, meaning that some can move further and some can eliminate more units during a single turn. A unit can only attack another unit when it has enough AP and the target is in range and 'in line of sight' where no other units or obstacles can be in between the attacker and their target.

Unit	**Description**	**MP cost**	**AP cost**	**Range**
Melee	Fast and close-range unit	1	1	1
Blaster	Standard mid-range unit	2	1	3
Sniper	Standard long-range unit	2	2	10
Laser	Slowest unit with the longest range	3	3	25

Table 2.1: The traits each unit possesses in the game

Resource management and positioning are the two key elements that make this game strategic. A player will have to distribute MP between their units to keep them out of the enemy's range and line of sight while also positioning their units to attack the enemy from a safe location. This isn't always going to be possible, and therefore the constraints of the MP and AP mechanics mean that sacrifices have to be made each turn in order to win. This game will be suitable for answering the first and second research questions listed in section 1.3.4 because moving and attacking can be directly linked to MP and AP costs allowing the A* algorithm to decide to move and attack units based on both positional and game-specific data.

Figure 2.2 shows the strategy game's structure and how the main *Game* class is derived from the *Scene* so that it can use functions such as *update* and *render* to integrate back to the core program in section 2.1.2. There are several important things about this design that should be explained further, one being the *Game* class contains a lot of static functions

such as *readMap* and *takeAction*, these functions are pure and do not rely on the current state of the game which is why they accept the data they operate on as a parameter rather than taking it from the instance of *Game*. These functions are easier to reason about and can be safely used in any context without the state of *Game* being touched — this allows A* to make use of these functions to create, operate on and evaluate states of the game without making destructive changes.

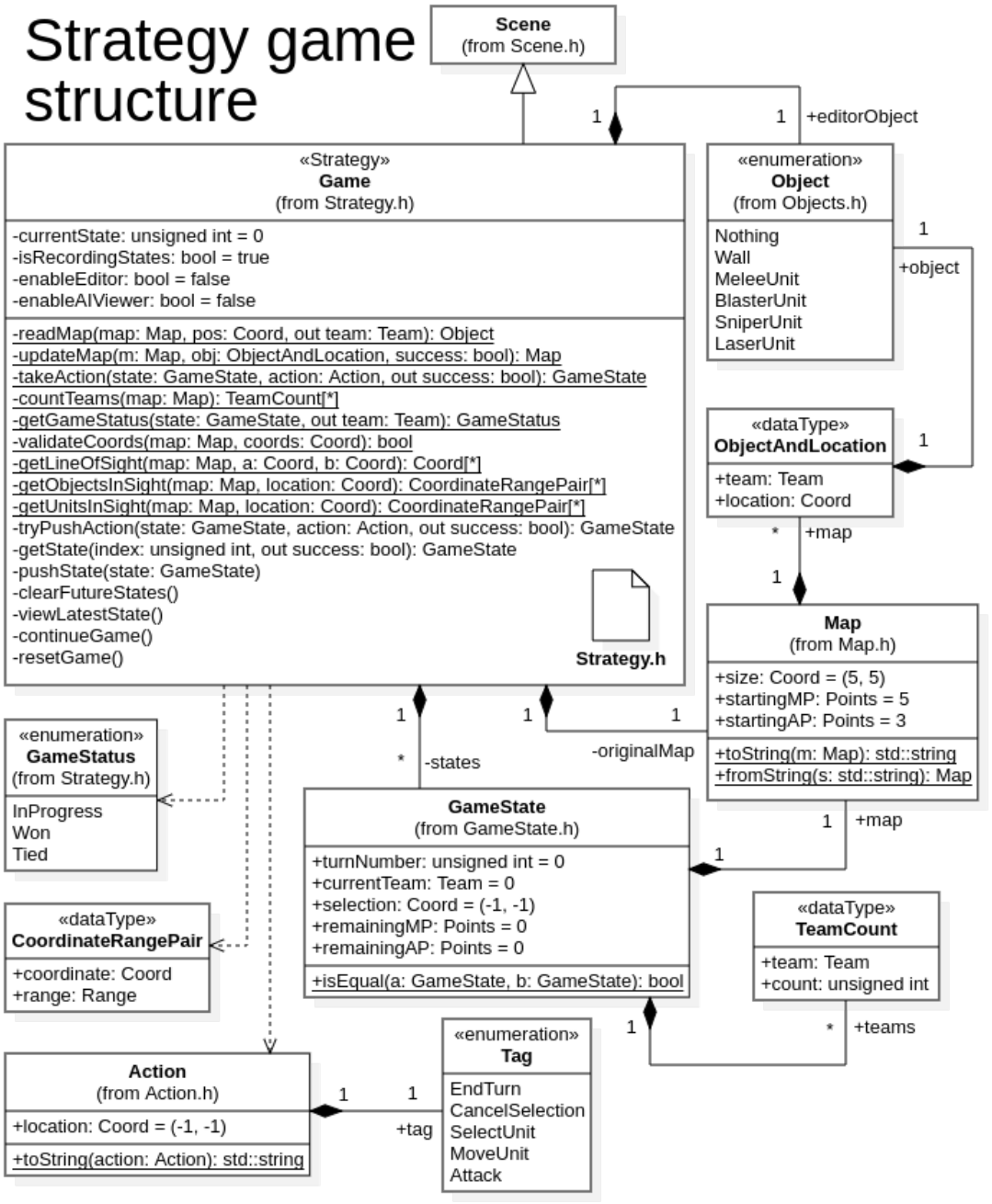

Figure 2.2: A UML diagram showing the implementation of the strategy game

The actual state of the game is contained in the class *GameState*. This is done for two

reasons, the first reason being that these states can be collected and revisited as needed for the final requirement of section 2.1.1. The map, the currently playing team, the turn number and number of resources remaining are all contained within the *GameState* itself and not in *Game*; the game then uses the selected *GameState's* contents to visualise the current state of the game on screen, making the viewing and overwriting of previous states as easy as viewing and manipulating this collection. The second reason is that the A* algorithm will process each *GameState* as a node in the decision making process and the *Actions* the players take will transform one state into a new, separate state to then be reconsidered for new *Actions*. The *GameState* class should represent what the AI's state and its perception of the game and be derived from the current state of the game, but because players have a complete set of information about the current state of the game, the *GameState* class serves as both the AI's current state as well as the actual state of the game.

Figure 2.3 shows a flowchart of how a turn will be played out and will be introduced in the function *continueGame* in the *Game* class shown figure 2.2. All controllers will return a stack of actions for the game loop to unwind and attempt to perform; this approach limits controllers to legitimate methods for working with *GameStates* and that *Game* can validate the decisions a controller makes and accept only those that are legal. The requirement of allowing a human to intervene section from section 2.1.1 is also considered — execution is will be interrupted during a human controller's turn and will resume when their turn ends. This process recursively fetches controllers' decisions and updates the collection of states until the game has concluded.

2.1.4 Designing tools for investigation

The requirements of a map editor and some way of browsing through the collection of states will be satisfied by using ImGui (Cornut, 2014) as stated in section 1.3.3. Both the editor and state browser will be contained in their own windows so that the map editor can be hidden when not needed, however the state browser will always be on display because of how important it is for observation of the AIs.

Figure 2.4 shows the four main parts to the map editor that are explained below:

- **Saving and loading:** This can be implemented with a text input for entering the name of a file combined with two buttons for handling saving and loading respec-

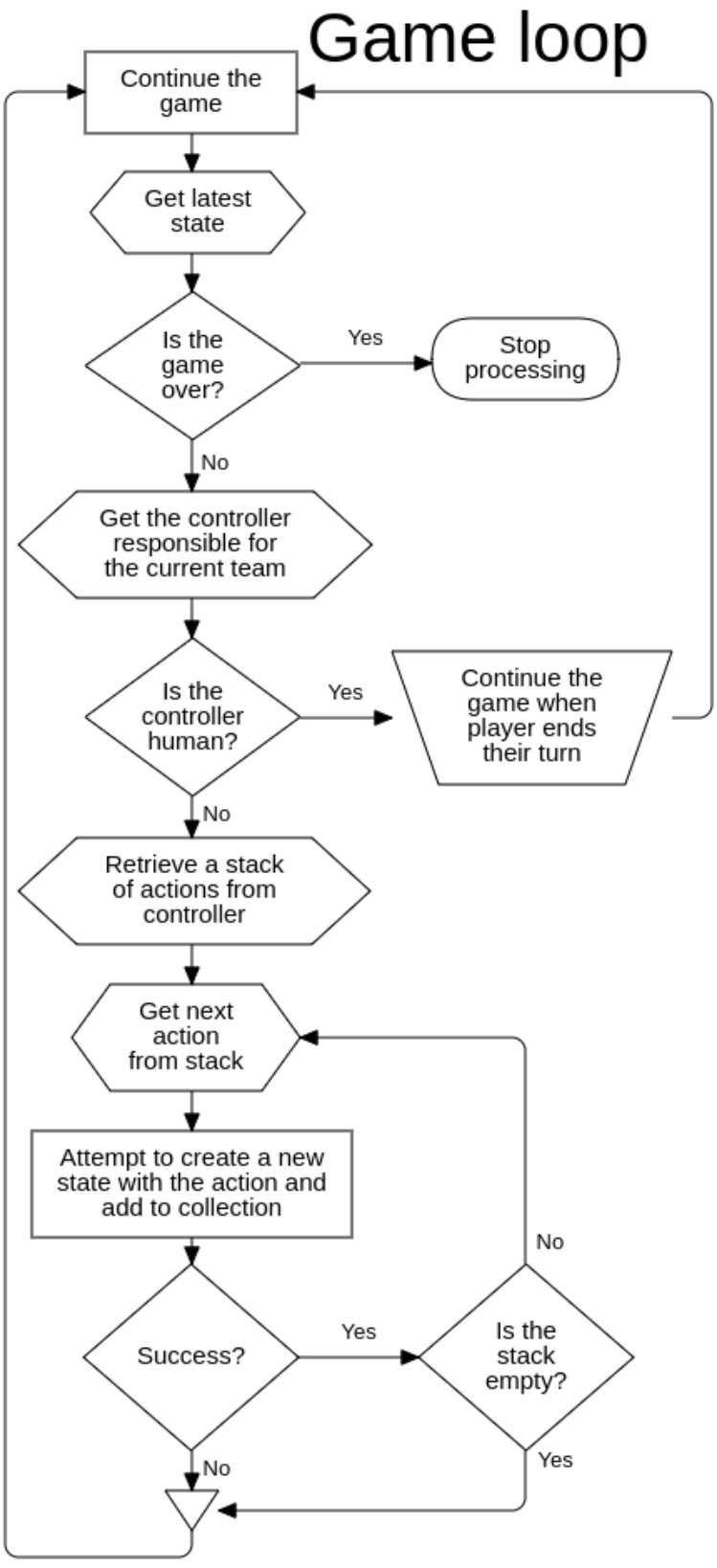

Figure 2.3: A flowchart showing how a turn is executed

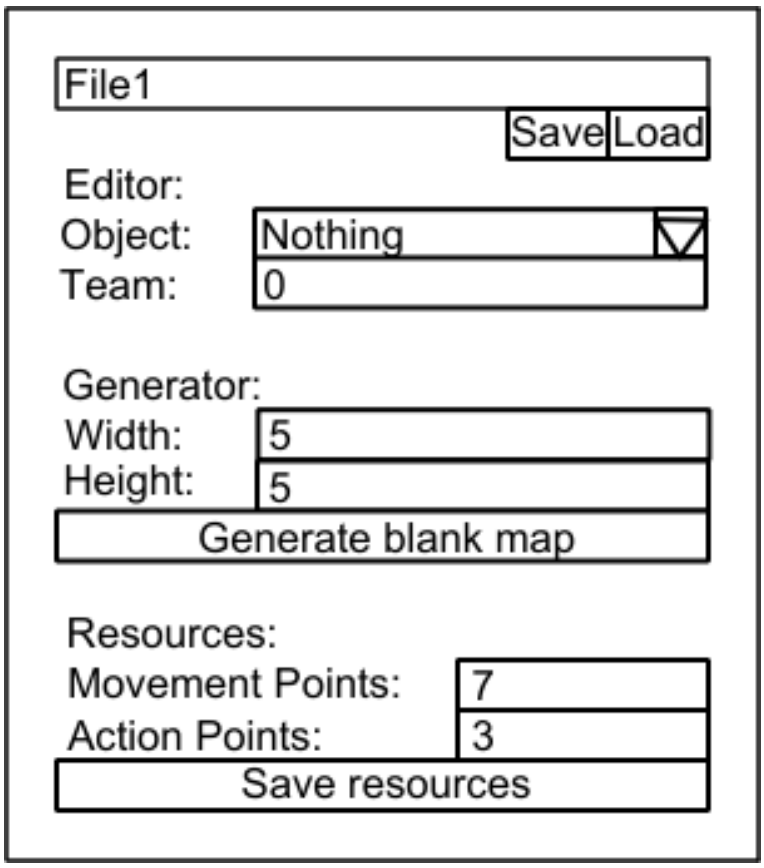

Figure 2.4: A draft of the design for the map editor window

tively.

- **Creating and deleting objects:** As shown in figure 2.2, an *Object* is just an enum that can represent nothing, a wall or a unit. When paired with a *Team* (represented as an integer value), a controller will then be assigned to that unit. This means that this part of the editor needs a drop-down box to select an *Object* and a number entry for specifying the *Team*. When the map editor window is open, left clicking and right clicking on the map will create and destroy objects.

- **Recreating maps:** This can be achieved with a number entries for specifying the width and height and a button that recreates the map when clicked.

- **Changing AP and MP:** Since changing the MP and AP requires the creation of a new *GameState*, the only way to change these values is to push a duplicate of the current state with the new resources. This can be also be done after a new map has been generated before saving to set the MP and AP allocated per turn.

Figure 2.5 shows how the state browser window should look. A simple scroll bar will suffice as a way of searching the collection — the variable *currentState* in *Game* as shown

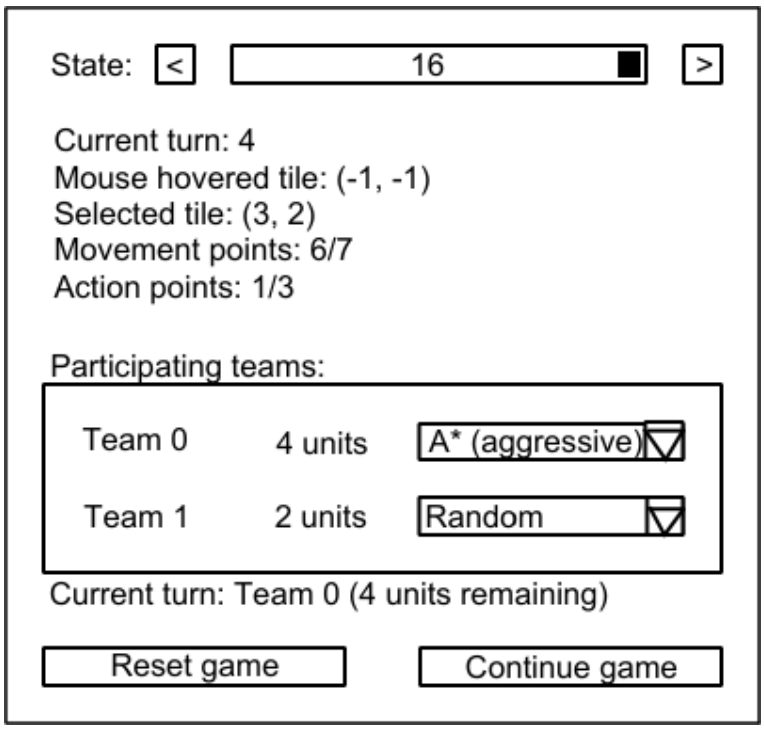

Figure 2.5: A draft of the design for the state browser

in figure 2.2 is an index referring to which state in the collection *states* to render on screen, so a scroll bar representing a range of values from zero to the size of *states* creates an easy way of examine all states of the game. This window also displays information about the state, such as the amount of MP and AP as well as the number of members in each team. It makes sense to also include the ability to change the controller for each team since each team is already being displayed, this will allow the alteration of which algorithm is controlling which team at any given moment. Finally, the ability to reset and continue the game, clearing the state collection and starting the process illustrated in figure 2.3 respectively, will be introduced with the 'reset game' and 'continue game' buttons that can be seen at the bottom of the window.

TODO: Talk about AI viewer, and storing different AIs based on team and controller

2.1.5 Designing the process of investigation

With the majority of the requirements in section 2.1.1 designed, the only one unsatisfied is the actual implementation of the A* algorithm — this will be programmed generically and be tuned for each use case during experimentation, with each variation of the algorithm being derived from interesting parts of previous iterations. The components of A* each variation is going to change is listed in table 2.2; the changes made will be decided based

Component	Description	Exemplary Changes
Goal function	Decide whether a given *GameState* should be considered a goal	Making ending the turn or eliminating at least one enemy a goal to satisfy
Cost implementation	Represent and store the cost of performing an *Action*	Separate the *Cost* type into a set of different elements using various traits and aspects of *Actions*
Weighing function	Generate the suitable *Cost* value for a given *Action*	Changing how *Actions* are valued to see how the AI's preferences affect behaviour
Heuristic function	Determine an order in which *GameStates* are explored	Don't guide it (therefore using Dijkstra's algorithm) or try different ways of estimating the *Cost* of satisfying the goal
Cost comparison function	Determine which *Cost* is preferable to another	Experiment with multiplying and ignoring elements of *Cost* as an alternative way of changing the AI's preferences

Table 2.2: The components of the A* algorithm and some examples of how they can be changed

on the studies of the other variations. For example, if a flaw in one variation's approach can be determined based on its observed behaviour, the next iteration might be adapted in an attempt to improve on the current approach.

TODO: Mention how cases will be stored somewhere?

2.2 Development

2.2.1 Milestone 1 — Creating the application's foundations

The first focus of development was to render 2D graphics, handle input, and to implement the *Scene* class to allow for each self-containing sandbox to be integrated. Starting with *main.cpp* and *App.h*, there needs to be a way for the application to be set up and for the main loop to be started. Following figure 2.1, the main loop is implemented in the function *start*, so *initialise* needs to request for *Resources* to *load* everything in the asset directory. The *initialise* function also opens the window and grants access for ImGui to render in that window. ImGui comes with a demo feature, so the *Console* class was also introduced so that error and debug messages can be displayed on-screen without needing a separate terminal.

Next was the *start* function which keeps the application in a loop until it needs to close.

From this loop, *deltatime* will be calculated and given to *update*, which doesn't do much other than store the location of the mouse cursor each frame for easy access. After *update*, any input detected by SFML gets polled and passed to the *handleEvent* function where it is decided which class should receive the event — if the user has clicked on an ImGui window, all text input should be given to ImGui so that they don't accidentally interact with games in the application. Finally, the *render* function is called which is responsible for clearing the screen, telling ImGui to render, and displaying that has been drawn onto the screen. A master debug menu will be displayed when *App*'s *debug* flag is set to *true*. This menu will show various, generic information about the application such as the frames per second and the cursor location. Additionally, it is through this menu that *Scenes* can be swapped to and their scene-specific debug windows can be shown and hidden.

After the application successfully looped, all that was needed was to implement the *Scene* class and to call functions *showScene*, *hideScene*, *onUpdate*, *onEvent* and *onRender* on *App*'s current *Scene* from their *App* counterparts. The function *switchScene* is introduced to *App* so that the beginning, showing and hiding of *Scenes* can be done automatically; this function accepts a string and will query *Resources* for a pointer to the relevant *Scene*. *Scenes* can also implement *addDebugMenuEntries* and *addDebugDetails* to add functionality to the debugging interface mentioned above. In order to test that the *App* can be started, switched to a *Scene*, and interacted with, the welcome *Scene* shown in figure 2.6 was made which featured a white circle that followed a mouse and an ImGui window specific to the *Scene*.

2.2.2 Milestone 2 — Building a sample game

Before the A* algorithm can be implemented for experimentation, there needs to be something in place for it to operate on and process. Since this project is being developed in an agile way, it was decided that a small, well-known and easily programmable game would be used to get a greater understanding of the requirements for this approach to game AI in case they need to be changed later. Tic-tac-toe fit this criteria well and served as a way of gauging the plausibility of creating game AI using A*. This game would also make use of a *Controller* class so that trivial AI, complex AI, and humans can play the game, allowing the implementation of A* to easily be tested against a human as it was being built. The human *Controller* was created first so that the rules of tic-tac-toe could be tested before an AI was introduced. The random team *Controller* mentioned in section

Figure 2.6: A screenshot of the welcome scene

2.1.1 came next: it chooses randomly from all available actions until it's turn is over. The idle *Controller* cannot be implemented for tic-tac-toe as turns cannot be skipped.

It is logical to build the tic-tac-toe game in the same way as the final game so that designs and implementations can be tested early, and so the tic-tac-toe game will make use of a *GameState* class so that turns can be undone and replayed. A simplified *GameState* viewer has been created with ImGui so that it can be tested as well — if appropriate, parts of this tool would be reused later to save time when developing the final version of the game. Figure 2.7 shows the tic-tac-toe game with fully functioning human *Controllers* and *GameState* viewer.

2.2.3 Milestone 3 — Implementing and allowing A* to play a game

With the simple game built in section 2.2.2, it was time to re-engineer the A* algorithm to operate on non-spatial data and play tic-tac-toe. This generic implementation of A* is shown in appendix A — the algorithm remains the same as the standard A* with a one exception: the types of the nodes, edges and the costs they evaluate to were replaced with

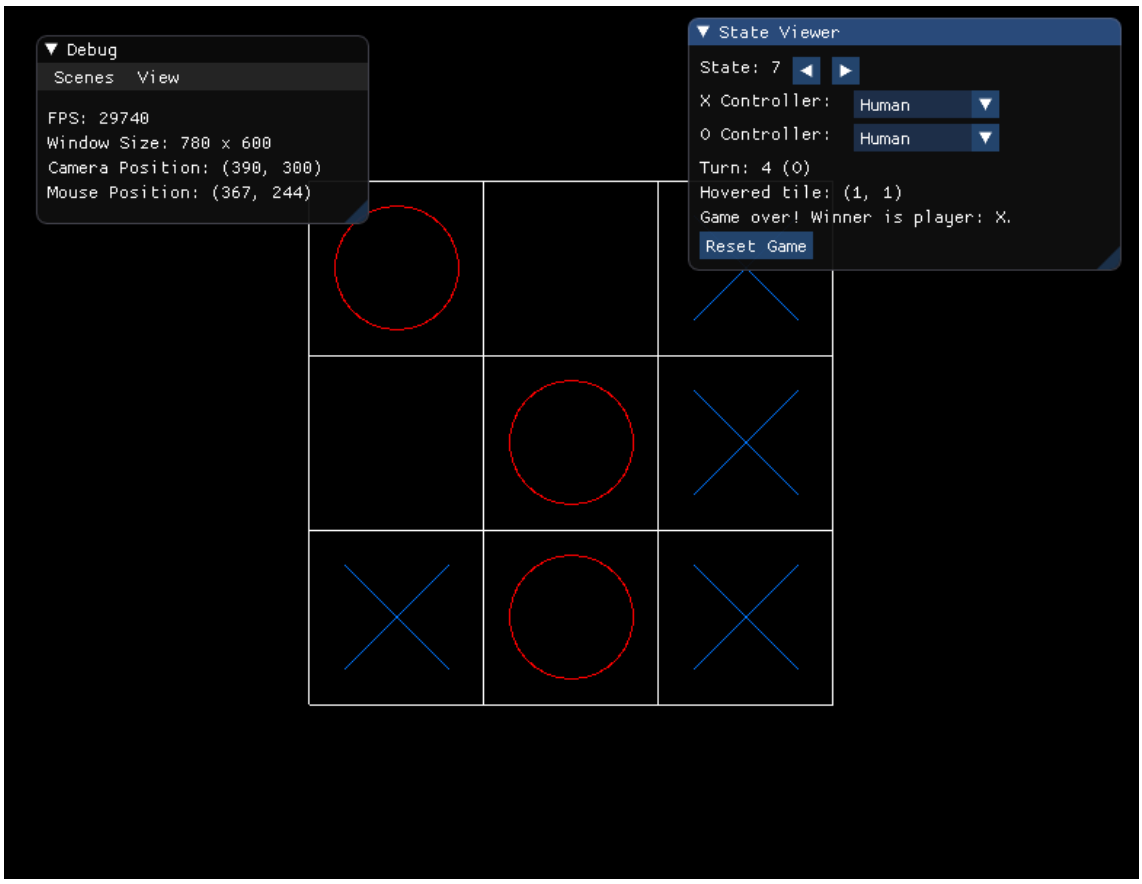

Figure 2.7: A screenshot of the tic-tac-toe game

the C++ templates S, A and C respectively. Doing this meant that the algorithm had further requirements, as there was no longer any way to calculate the distance between two instances of S, or any way of expanding nodes. Function pointers, or alternatively a functor could be used, solve this by requiring this functionality being passed in when the algorithm is used. For instance, if used for pathfinding, this algorithm would be given a function that would find adjacent tiles or waypoints to a given node so that nodes can be expanded properly. The types and functions supplied for the AI to play tic-tac-toe are listed below:

- **States:** The states of the game are represented by the tic-tac-toe *GameState* class.
- **Actions:** A move is made by placing a mark in the grid, so an action is just a location represented by a pair of integers (the *sf::Vector2i* class).
- **Costs:** An integer value represents the penalties for making illogical moves.
- **Goal nodes:** Every action was valid as long as a mark was placed on an empty space, therefore the AI wasn't directed towards a specific goal but rather to just

make a move.

- **Node expansion:** Took a *GameState* and finds empty spaces for marks to go to return a collection of actions and the *GameStates* they led to.
- **Cost comparison:** Costs were represented as an integer, so finding the lesser value was trivial and the standard $<$ operator was used.
- **Heuristic function:** No heuristic was supplied — $h(n)$ returned the same value and therefore A* turned into Dijkstra's algorithm for this test.
- **Weighing function:** Penalties were applied in accordance to how inappropriate a move was. Moves which didn't finish possible game-winning rows of three are heavily penalised to ensure that the AI would win when it can. Similarly, moves which didn't block two marks in a row owned by the opponent were severely penalised to incentivise blocking the opponent's obvious attempts of winning. Other than that, the penalty of a move would depend on how many 'near-wins' are created for the AI and how many are left unchecked by the opponent. This is an arbitrary function and will be scrutinised during the actual experiment.

While the AI did play the game successfully, it didn't prove whether game AI using A* was actually plausible. Each turn in a game of tic-tac-toe consists of exactly one move, and so the A* algorithm was creating a sequence containing a single action — it compared all possible moves and chose the option that was penalised the least by the weighing function. From this, more requirements for the experiment could be identified; a game where the AI can do multiple actions per turn would provide results with greater detail and would be a more realistic use case. The requirements in section 2.1.1 and the design of the game in section 2.1.3 were changed to reflect this realisation. Additionally, since this game is going to feature a lot more states than tic-tac-toe and the only way of revisiting previous states was to click the arrow buttons shown in figure 2.7, the design of the state viewer in section 2.1.4 was changed to include a scrollbar for faster browsing of states.

2.2.4 Milestone 4 — Building the final game

After programming and testing the A* algorithm, the final piece before the case studies could be started was the game itself. The game designed in section 2.1.3 was built from the ground up to support both human and non-human players; visuals on the screen

ensured that information such as sight lines and the resources remaining was properly communicated regardless of state. Units were represented by letters and their colour shows which team they are on. Additionally, the generic implementation of A* was used for standard pathfinding, so that players didn't have to move their units step by step. Figure 2.8 shows the finished game in action from a human player's perspective, showcasing the pathfinding of the melee unit in conjunction with the resource cost for performing the manoeuvre.

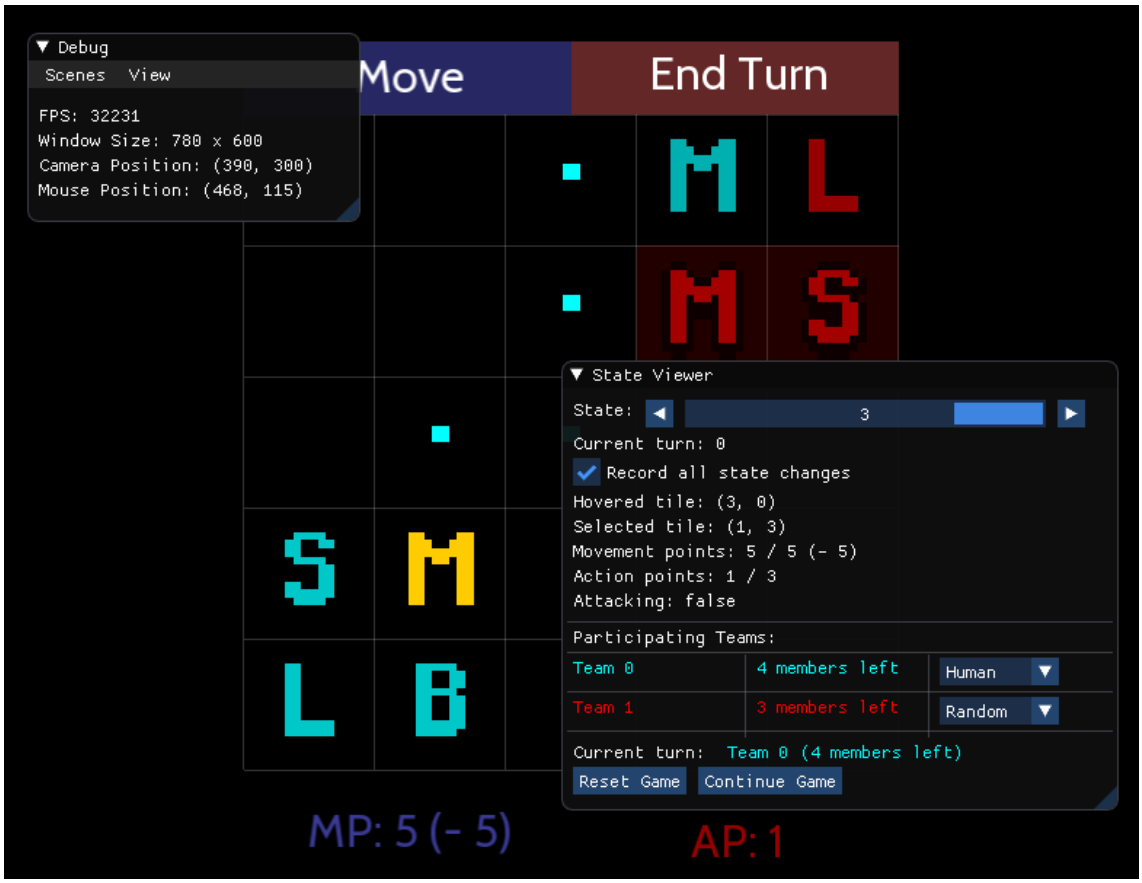

Figure 2.8: A screenshot of the strategy game showing a turn in progress

The state viewer and information window created in section 2.2.2 was adjusted to show details about the new game, as well as being expanded to handle multiple teams playing a single game. Finally, the map editor as well as the ability to save and load maps were introduced. Overall, development went smoothly — having the tic-tac-toe and strategy games share similar approaches using *GameStates* and a variety of static functions meant that development was a matter of adapting and expanding previously written functions to suit the new game. With the human and trivial controllers properly set up to play the game, the case study could finally begin.

2.3 Results and Evaluation (11–12 pages including critical review)

2.3.1 Case study one

Configuration

- **Goal:** A *GameState* is considered a goal node when the currently playing team changes, meaning that the previous *Action* was the ending of the current turn and give control to the next team.
- **Cost:** Represented by a set of integer values representing different penalties: enemies remaining, allies lost and allies at risk. The aim is to make the AI select actions that eliminate opposing units without exposing allies. Friendly fire is also an option and actions that kill an allied unit are given penalties. Summing *Cost* values is achieved by adding the components together, resulting in a *Cost* value containing totals for each penalty.
- **Cost comparison:** A *Personality* class has been introduced which contains floating point values as multipliers for the elements of *Cost*. When comparing two *Cost* instances, their values are manipulated by a *Personality* before being summed into one value, it is these totals that are then used in comparisons.
- **Heuristic function:** Nothing useful given — only a *Cost* value containing zero for each penalty is given.
- **Weighing function:** Each element of *Cost* is found through comparison of *GameStates*. Constant arbitrary values are used to create the penalties; for example, 10 points are added to the 'enemies remaining' penalty total. Since selecting units or ending the turn doesn't require or change anything, ending the turn isn't penalised.

This approach is very similar to the way the AI in tic-tac-toe compared different moves. No goal or heuristic was provided, so the AI relied entirely on the comparison of the *Costs* for performing *Actions*. The notion of a *Personality* was introduced to tweak the AI's values; perhaps it is desired for the AI to not care about killing friendly units if it means winning. Ending the turn doesn't award any penalties because the *Cost* of the *Action* sequence comes from what the AI does, and since ending the turn does nothing to the game there's no reason to add a penalty for it.

Observations

The AI displayed no behaviour regardless of which map from appendix B was selected, although it still managed to win due to the random controller killing its own units. It ended its turn straight away much like one of the trivial controllers. This reinforces how poor tic-tac-toe was for experimentation; the lack of depth meant that this flaw remained unexposed. As discussed in 1.2.5, A* uses non-negative cost values so that higher-costed action sequences can be deferred. With every move being penalised on some scale, performing two *Actions* in a row will typically cost more than performing a single *Action* even if the resulting *GameState* is more favourable. Since there wasn't a non-zero *Cost* for ending the turn, there wasn't a combination of *Actions* that was preferable to doing nothing.

This case illustrates one flaw of developing game AI in this way: there must be extra consideration for possibly numerous areas of the AI in order to ensure that the behaviour is within expected parameters. Further cases need some way of deferring the ending of the turn.

2.3.2 Case study two

Configuration

- **Goal:** End the current turn.
- **Cost:** A collection of integer values as described in case study one, but with additional penalties for wasting MP and AP.
- **Cost comparison:** Same method of comparison as case study one.
- **Heuristic function:** Nothing useful given — only a *Cost* value containing zero for each penalty is given.
- **Weighing function:** Same as case study one with one exception. The end turn *Action* applies a heavy penalty based on how much MP and AP is remaining.

This case is almost identical to case study one from section 2.3.1; in that case study, the AI chose to do nothing every time because the end turn *Action* was weighed to have zero *Cost*. This has been changed for this case, where ending the turn now incurs a penalty depending on how much MP and AP is remaining in an attempt to incentivise performing *Actions*

and spending these resources to lower the penalty. One final thing that was considered was the multiplying all elements of the *Cost* value when the turn is ended. This is so that additive and multiplicative penalties can be tested in case one is more effective than the other. The AI viewer will allow the manipulation of these values so that different behaviours can be found during play.

Observations

Initially, when playing on the map B.1 the AI didn't move at all and repeated the behaviour of case study one in section 2.3.1. After manipulating the multiplier for MP and AP penalties, the AI was quickly compelled to win the game against both the idle and random team controllers. This is because the penalty for wasting MP and AP becomes great enough that ending a turn with less resources and killing more enemies is favourable to doing nothing, which is promising. It doesn't matter how high these penalties were set; when the same amount of resources have been spent, the AI will take the other penalties into consideration to find the absolute minimum.

Increasing the total turn multiplier from one slowed down the turn dramatically, becoming impractical at multipliers higher than 3. It is suspected that this is because A* cannot terminate computation until it reaches a goal node or it runs out of open nodes, and doing this postpones the evaluation of all end turn *Actions* until every other *Action* has been processed. This is unhelpful and inefficient.

This AI heavily struggled with map B.2. The AI took a lot longer to win the game in real time, taking almost 10 minutes with the MP and AP multipliers set to 5 and the end turn multiplier set to 1. This demonstrates another flaw in using penalties to influence AI — A*, or in this case, Dijkstra's algorithm, was constantly evaluating nodes with the lowest *Cost*, and so because there was no way of ending the current turn without incurring a large penalty, the algorithm had to process the majority of possibilities before an end turn *Action* would be considered the next thing to process. Multiple sets of values were tested, but a solution to speeding up the AI could not be found. Further cases will need to consider the following:

- Using a heuristic function to perform best-first searches, which could be difficult because 'best' is subjective when it comes to gameplay.
- Limiting the depth of the search, although there needs to be some sort of backup in

place. If an end turn *Action* was already found it would have been used, therefore there wouldn't be any backups other than ending the turn doing nothing, which is another problem.

- Changing penalties in some way so that the AI isn't penalised for performing *Actions* in a bad situation.

END

- This area can be further subdivided

- Give summary of results in this case, results will be how the AI behaves in each case - Evaluate through comparison between other products and other case studies - Explain findings, especially if unexpected - Discuss further work if applicable - Demonstrate achievements and indicate how the results meets project requirements - Evaluate against criteria with support of gathered data - Assess meaning of results of testing - If a criteria has been failed, assess why it did not happen - Was the criteria faulty? If so, explain how knowledge of the problem domain has changed - How does what I did prove the criteria?

Chapter 3

Conclusions

3.1 Conclusions and Critical Review

- Compulsory chapter - (2–3 pages)

- Evaluate to what extent the project objectives have been fulfilled - Reflect on the project as a whole - Describe limitations - Propose a set of recommendations for improving the methods used - Give suggestions for further work - 'Step away' from the project and assess learning experience - If I had to do it again and I knew at the beginning what I know now, what would be different?

3.2 Conclusions and recommendations

- This section should briefly summarise: - The project objectives - Findings - Methods used - Deliverable developed - Evaluation of the quality of the product - Usefulness of the findings - Worth of learning experience

Bibliography

Bass, L., Weber, I., & Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.

Baxter, P., & Jack, S. (2008). Qualitative case study methodology: Study design and implementation for novice researchers. *The qualitative report*, *13*(4), 544–559.

Blow, J. (2004). Game development: Harder than you think. *Queue*, *1*(10), 28.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of game development*, *1*(1), 7–28.

Buro, M. (2004). Call for AI research in RTS games. In *Proceedings of the AAAI-04 Workshop on Challenges in Game AI* (pp. 139–142). AAAI press.

Colledanchise, M., Marzinotto, A., & Ögren, P. (2014). Performance analysis of stochastic behavior trees. In *2014 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 3265–3272). IEEE.

Cornut, O. (2014). Immediate mode GUI (ImGui) (Version 1.74). Retrieved December 28, 2019, from https://github.com/ocurnut/imgui

Cui, X., & Shi, H. (2011). A*-based pathfinding in modern computer games. *International Journal of Computer Science and Network Security*, *11*(1), 125–130.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269–271.

Diller, D. E., Ferguson, W., Leung, A. M., Benyo, B., & Foley, D. (2004). Behavior modeling in commercial games. In *Proceedings of the 2004 Conference on Behavior Representation in Modeling and Simulation (BRIMS)* (pp. 17–20).

Epic Games. (1998). Unreal Engine (Version 4.23.1). Retrieved November 30, 2019, from https://www.unrealengine.com/en-US/

Forbus, K. D., Mahoney, J. V., & Dill, K. (2002). How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, *17*(4), 25–30.

Fowler, M., Highsmith, J. et al. (2001). The agile manifesto. *Software Development*, *9*(8), 28–35.

Friedman, J. H., Bentley, J. L., & Finkel, R. A. (1976). An algorithm for finding best matches in logarithmic time. *ACM Trans. Math. Software*, *3*(SLAC-PUB-1549-REV. 2), 209–226.

Gomilia, L. (2007). Simple and Fast Multimedia Library (SFML) (Version 2.5.1). Retrieved December 28, 2019, from https://sfml-dev.org

Graham, R., McCabe, H., & Sheridan, S. (2003). Pathfinding in computer games. *The ITB Journal*, *4*(2), 6.

Harmon, V. (2002). An economic approach to goal-directed reasoning in an RTS. *AI Game Programming Wisdom*, 402–410.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, *4*(2), 100–107.

Higgins, D. (2002). Generic A* pathfinding. *AI Game Programming Wisdom*, 114–121.

Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, *27*(1), 97–109.

Laird, J., & VanLent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI magazine*, *22*(2), 15–15.

Lantinga, S. (1998). Simple DirectMedia Layer (SDL) (Version 2.0.10). Retrieved December 28, 2019, from https://libsdl.org

Leigh, R., Louis, S. J., & Miles, C. (2007). Using a genetic algorithm to explore A*-like pathfinding algorithms. In *2007 IEEE Symposium on Computational Intelligence and Games* (pp. 72–79). IEEE.

Lester, P. (2005). A* pathfinding for beginners. Retrieved November 30, 2019, from http://www.gamedev.net/reference/articles/article2003.asp

Lim, C.-U., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game DEFCON. In *European Conference on the Applications of Evolutionary Computation* (pp. 100–110). Springer.

Linietsky, J., & Manzur, A. (2014). Godot (Version 3.1.2). Retrieved December 28, 2019, from https://godotengine.org

Martin, J. (1991). *Rapid application development*. Macmillan Publishing Co., Inc.

Mettke, M. (2015). Nuklear. Retrieved December 28, 2019, from https://github.com/immediate-mode-ui/nuklear

Millington, I. (2019). *AI for Games*. CRC Press.

Monolith Productions. (2005). F.E.A.R.

Nareyek, A. (2004). AI in computer games. *Queue*, *1*(10), 58.

Newzoo. (2019). *2019 Global Games Market per device and segment*. Retrieved November 30, 2019, from https://newzoo.com/key-numbers/

Orkin, J. (2003). Applying goal-oriented action planning to games. *AI game programming wisdom*, *2*, 217–228.

Orkin, J. (2006). Three states and a plan: the AI of FEAR. In *Game Developers Conference* (Vol. 2006, p. 4).

Pearl, J. (1984). Heuristics: Intelligent search strategies for computer problem solving.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering* (pp. 328–338). IEEE Computer Society Press.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence: A modern approach*. Malaysia; Pearson Education Limited.

Scott, B. (2002). The illusion of intelligence. *AI game programming wisdom*, *1*, 16–20.

Shoulson, A., Garcia, F. M., Jones, M., Mead, R., & Badler, N. I. (2011). Parameterizing behavior trees. In *International Conference on Motion in Games* (pp. 144–155). Springer.

Silicon Graphics. (1992). Open Graphics Library (OpenGL) (Version 4.7). Retrieved December 28, 2019, from https://opengl.org

Stake, R. E. (1995). *The art of case study research*. Sage.

Stanciu, P.-L., & PETRUŞEL, R. (2012). Implementing Recommendation Algorithms for Decision Making Processes. *Informatica Economica*, *16*(3).

Stentz, A. (1996). Map-based strategies for robot navigation in unknown environments. In *AAAI spring symposium on planning with incomplete information for robot problems* (pp. 110–116).

Stentz, A. (1997). Optimal and efficient path planning for partially known environments. In *Intelligent Unmanned Ground Vehicles* (pp. 203–220). Springer.

Sweetser, P., & Wiles, J. (2002). Current AI in games: A review. *Australian Journal of Intelligent Information Processing Systems*, *8*(1), 24–42.

Tozour, P. (2002). The evolution of game ai. *AI game programming wisdom*, *1*, 3–15.

Unity Technologies. (2005). Unity Engine (Version 2019.2.14). Retrieved November 30, 2019, from https://unity.com

Weber, B. G., Mateas, M., & Jhala, A. (2011). Building human-level ai for real-time strategy games. In *2011 AAAI Fall Symposium Series*.

Yap, P. (2002). Grid-based path-finding. In *Conference of the Canadian Society for Computational Studies of Intelligence* (pp. 44–55). Springer.

Yin, R. (2003). K.(2003). Case study research: Design and methods. *Sage Publications, Inc*, *5*, 11.

Appendices

Appendix A

Generic A* implementation

```
1   // A functor for using AStar
2   // Templates: thought STATE, ACTION, decision COST
3   template <class S, class A, class C>
4   struct AStar {
5
6   public:
7
8     // Public getters
9     const std::pair<A, C>& getCurrentAction() const { return
        ↪ currentAction; }
10    const unsigned int& getStatesProcessed() const { return
        ↪ statesProcessed; }
11    const std::vector<S>& getRemaining() const { return
        ↪ remaining; }
12    const std::unordered_map<S, C>& getFScores() const { return
        ↪ fScore; }
13    const std::unordered_map<S, C>& getGScores() const { return
        ↪ gScore; }
14
15    // Evaluates options and returns a stack of actions to take
16    std::pair<bool, std::stack<A>> operator() (
17        const S& startingState,
18        const C& minimumCost,
19        const C& maximumCost,
20        std::function<std::vector<A>(const S&)>
            ↪ getPossibleActions,
21        std::function<bool(const S&, const S&)> isStateEndpoint,
22        std::function<C(const S&)> heuristic,
23        std::function<C(const S&, const S&, const S&, const A&)>
            ↪  weighAction,
24        std::function<std::pair<bool, const S>(const S&, const A
            ↪ &)> takeAction,
25        std::function<bool(const C&, const C&)> compareCost =
            ↪ std::less<C>()) {
26
27      // All available states to explore
28      remaining = {startingState};
29
30      // Keep track of states already evaluated
31      std::vector<S> evaluated = {};
32
```

```
33          // Map of which action led to which thought state
34          history.clear();
35
36          // How much decision power gained so far in each state
37          gScore = { { startingState, minimumCost } };
38
39          // Map of predicted decision power from current state
40          fScore = { { startingState, heuristic(startingState) } };
41
42          // Keep track of the number of actions processed
43          statesProcessed = 0;
44
45          // Keep processing until there are no states left to check
46          while (remaining.size() > 0) {
47
48            // @ANALYSIS: Record how many moves have been processed
49            statesProcessed += 1;
50
51            // Get the highest priority state to operate on
52            auto current = std::min_element(remaining.begin(),
                ↪ remaining.end(),
53              [this, &maximumCost, &compareCost](const S& a, const
                  ↪ S& b) {
54
55              // Find fScores of a and b
56              const C& fA = fScore.find(a) != fScore.end() ?
57                  fScore[a] : maximumCost;
58              const C& fB = fScore.find(b) != fScore.end() ?
59                  fScore[b] : maximumCost;
60
61              // Return whether fA should come before fB
62              return compareCost(fA, fB);
63            });
64
65            // If we've arrived at a node that can be considered the
                ↪ goal, stop
66            const S state = *current;
67            if (isStateEndpoint(startingState, state)) {
68
69              // Reconstruct the processes taken to get here
70              S pathNode = state;
71              std::stack<A> actionsTaken;
72
73              // Build the path of actions from finish to start
74              while (pathNode != startingState) {
75
76                // Find how we got to the current state
77                const auto& it = history.find(pathNode);
78                if (it != history.end()) {
79
80                  // Get the data contained in the kvp
81                  const auto& previous = it->second;
82
83                  // Focus on the previous state for next iteration
84                  pathNode = previous.first;
85
86                  // Add action to the stack
87                  actionsTaken.push(previous.second);
88                }
89
```

```
90              // This shouldn't trigger as all routes should be
                   ↪ traceable
91              // However, if it does, stop the infinite loop
92              else {
93                assert(false);
94                return std::make_pair(false, actionsTaken);
95              }
96            }
97            return std::make_pair(true, actionsTaken);
98          }
99
100         // No longer consider the current state
101         evaluated.insert(evaluated.end(), state);
102         remaining.erase(current);
103
104         // Initialise gScore of state if it's not there
105         if (gScore.find(state) == gScore.end()) {
106           gScore[state] = maximumCost;
107         }
108
109         // Get possible (neighbour) states by trying all
                ↪ possible actions
110         const std::vector<A> actions = getPossibleActions(state)
                ↪ ;
111         std::vector<std::pair<S, A>> states;
112         std::for_each(actions.begin(), actions.end(),
113             [&states, &evaluated, &state, &takeAction]
114                 (const A& action) {
115
116           // Try taking the action with the current state
117           const auto& attempt = takeAction(state, action);
118
119           // Consider any valid states that haven't been
                  ↪ evaluated
120           if (attempt.first
121               && std::find(evaluated.begin(), evaluated.end(),
                      ↪ attempt.second)
122                   == evaluated.end()) {
123             states.insert(states.end(), std::make_pair(attempt.
                    ↪ second, action));
124           }
125         });
126
127         // For each neighbour
128         std::for_each(states.begin(), states.end(),
129             [this, &startingState, &state,
130             &maximumCost, &weighAction, &heuristic, &compareCost
                    ↪ ]
131                 (const std::pair<S, A>& future) {
132
133           // Initialise gScore of neighbour if it's not there
134           if (gScore.find(future.first) == gScore.end()) {
135             gScore[future.first] = maximumCost;
136           }
137
138           // Work out cost of taking this action with the
                  ↪ current state
139           const C tentative_gScore = gScore[state] +
140               weighAction(startingState, state, future.first,
                      ↪ future.second);
```

```
141
142             // @ANALYSIS: Record what the action is
143             currentAction = std::make_pair(future.second,
                    ↪ tentative_gScore);
144
145             // If our projected score is better than the one
                    ↪ recorded
146             if (compareCost(tentative_gScore, gScore[future.first
                    ↪ ])) {
147
148               // Record how we got to this node (insert new state
                      ↪ and the action)
149               history.insert(std::make_pair(
150                     future.first,
151                     std::make_pair(state, future.second)));
152
153               // Update the gScore to the lower score
154               gScore[future.first] = tentative_gScore;
155
156               // Update fScore to a more accurate representation
157               fScore[future.first] = tentative_gScore + heuristic(
                      ↪ future.first);
158
159               // If the current future isn't queued to be
                      ↪ evaluated next, add it
160               if (std::find(remaining.begin(), remaining.end(),
                      ↪ future.first)
161                    == remaining.end()) {
162                 remaining.insert(remaining.end(), future.first);
163               }
164             }
165           });
166       }
167
168       // Return unsuccessfully with the current state
169       return std::make_pair(false, std::stack<A>());
170     }
171
172   private:
173
174     // All available states to explore
175     std::vector<S> remaining;
176
177     // Store FScores (costs of finishing pathing of all states)
178     std::unordered_map<S, C> fScore;
179
180     // Accurate cost of getting to a state
181     std::unordered_map<S, C> gScore;
182
183     // Map of which action led to which thought state
184     // Map<future State, Pair<previous State, Action taken>>
185     std::unordered_map<S, std::pair<S, A>> history;
186
187     // Keep track of the number of actions processed
188     unsigned int statesProcessed = 0;
189
190     // Keep track of the current Action and its Cost
191     std::pair<A, C> currentAction;
192
193   };
```

```
194 }
195
196 #endif
```

Appendix B

Strategy game maps

B.1 Default 5x5 map

Figure B.1 shows the default map. 5x5 is big enough for basic play but small enough to keep processing fast. A good gauge of how well the AI performs is how well it copes with bigger map sizes, but 5x5 is the base requirement to be considered worth investigating. Teams on this map start with 5 MP and 3 AP per turn.

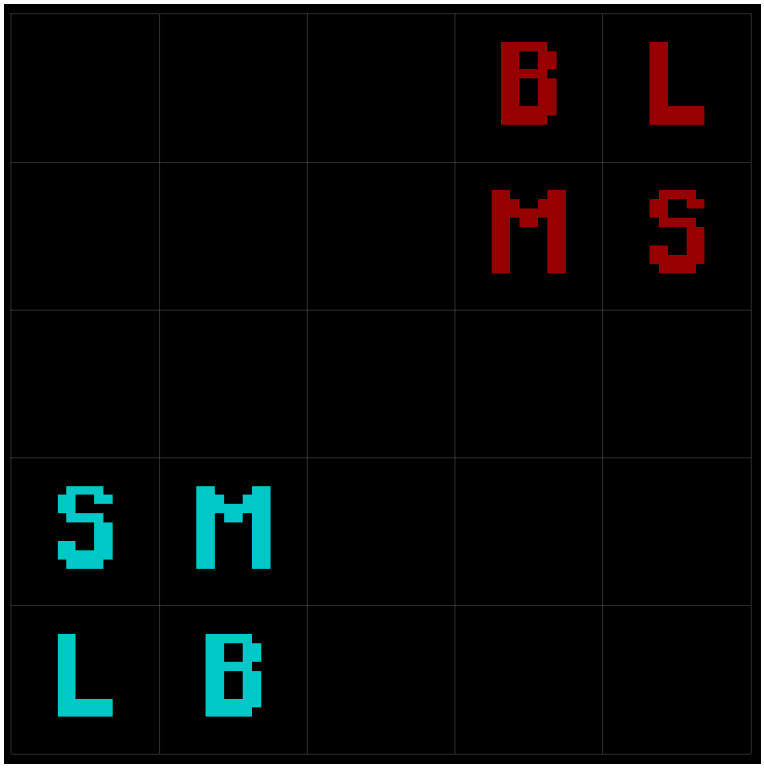

Figure B.1: A symmetrical square 5x5 map

B.2 Cross-wall 5x5 map

Figure B.2 shows the default map with an orthogonal cross placed in the center. This cross obstructs line of sight and the AI must attack the walls to remove them and attack the enemy. This map requires teams to do more *Actions* to win, and therefore requires multiple turns to be beaten. Teams on this map start with 5 MP and 3 AP per turn.

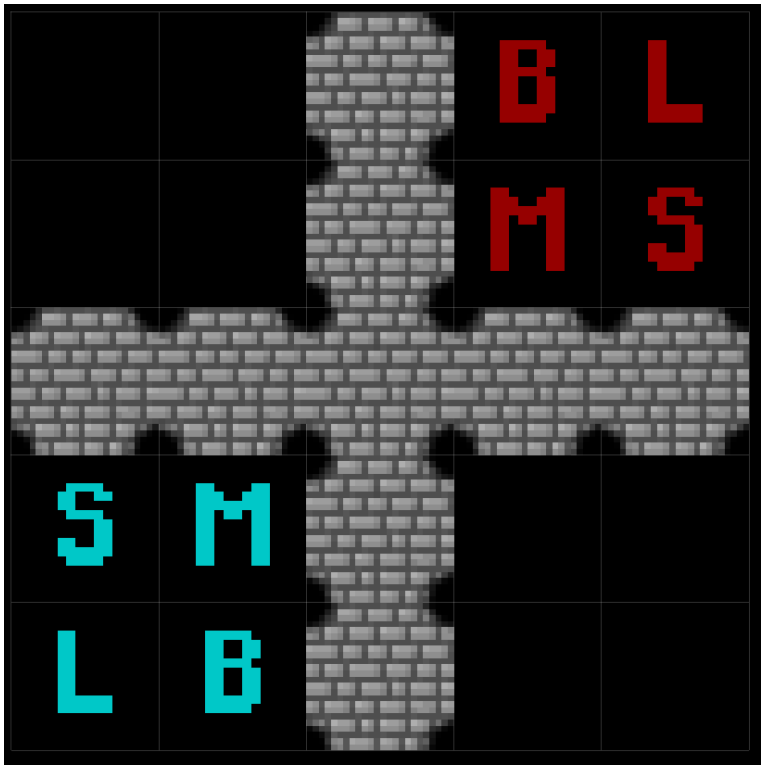

Figure B.2: A symmetrical square map with obstacles obstructing lines of sight