Interim Report: Multiplayer Strategy Game

Adam Boulton

11th December 2012

Abstract

This report researches the various problems implied in the creation of a multiplayer strategy game. It looks into the Java programming langauge, Pathfinding, Artificial Intelligence and also discusses a mathetmatical solution to checking visibility between two points on a grid. The report also suggests approaches for implementing the overall aims and objectives of the project.

Contents

List of Algorithms

1 Introduction

The goal of my final year project is to design and implement a multiplayer, turn based strategy game. Whilst the game will be playable by two competing players, it will also be playable against a customisable artificial intelligence opponent. The game's premise is to provide each player with a small selection of units on a two dimensional landscape. Each player takes in turn to select a single unit to move and attack with, and the game ends when one team is eliminated. There will be a small number of unit types, and each will be designed to have clear strength and weaknesses. This will make each unit type have a clear role for players to think about and be aware of, and provide tactical options for players and AI alike to exploit.

There are several problems that require solving to achieve a successful implementation of this project idea. Key problems include the initial implementation of the game's grid based structure: how the grid will be represented in Java, how each unit will be implemented, a way to find paths between locations on the grid, check moves are legal, provide a challenging AI player to compete against and also the rules and conditions of the game itself. The game will need to detect when a player has one, keep track of scores, and so on.

Pathfinding is a way of finding the optimum path between two points, avoiding any potential obstacles on the way. With a pathfinder, we look ahead to plan a path, rather than simply reacting to obstacles: if we attempt to take a direct route from one point to another and encounter an obstacle, we would then have to traverse round it. A pathfinder would never encounter the obstacle at all.[6]

Move legality is another key problem. Whilst one aspect of this could be solved through the pathfinder (simply have a maximum path length that matches a unit's maximum move distance), a more complex issue is checking whether one unit can actually see another when attacking. If the line of sight is blocked by a wall, we need to be able to detect this. A simplistic solution could simply be to limit shooting directions to pure diagonals or straight lines, but this could negatively impact gameplay and tactical options.

The final aspect is implementing a strong and interesting AI engine. The AI player would need to be able to provide a scaleable challenge for a human player, without using behaviours that people may consider unfair. An AI player should be affected by the same limitations a human player would also face. Difficulty should be scaleable through a number of factors: how many turns the engine can "think ahead", how likely it is to make mistakes, different heuristics to alter its play style between aggressive or defensive, and even conditional behaviours triggered by certain situations, such as holding a major unit advantage or disadvantage. Any tactic a human player could use, any potential flaw, the AI needs to have at its disposal in order to be as realistic as possible.

All of these problems require significant research and experimentation in order to achieve a high quality final product. In summary, the project features these overall aims and objectives:

- A two dimensional grid on which units can move around
  - There will be several different types of grid cells, some which will allow movement or shooting through, and some that will not
  - Users will be able to generate their own grid designs or play on a pre designed or saved one
  - Grid squares will be visually distinctive for ease of use
  - Players should be able to shoot at any angle and the game will calculate if the shot is blocked
- Several types of unit
  - Each type of unit needs to be balanced against the next - a unit that can cover a large distance in a move should not also be able to attack over long distances, and a unit that can only move a short distance should be able attack over long distances in order to compensate
- Multiplayer functionality
  - Two players should be able to take turns in order to attempt to defeat the other team's units
  - A score system should allow players to track scores and statistics: total wins, total moves, total units defeated, and so on
- Robust pathfinding
  - A pathfinding algorithm that always produces a valid move between two points. Moves will be restricted because some tiles will not allow units to pass through them, and each unit will have a maximum move distance, making some locations impossible to reach.
  - The pathfinder should be efficient and have a low computing cost
- Artificial intelligence
  - The AI should have multiple difficulty options that can be customised before matches, with a few presets such as an 'easy' or 'hard' setting
  - The AI should be able to think several moves ahead
  - The AI should have human traits such as a chance to make mistakes, or different levels of aggression depending on the current game situation
- Further features
  - Visually distinctive units and tiles
  - A user friendly interface

2 Background

2.1 The Java Language

Java is a high level, Object Oriented programming language create by Sun Microsystems (now a part of the Oracle Corporation). Java is a class based language operating on a 'write once, run anywhere' principle, meaning that its code can be run on any system that has the correct version of the Java Runtime Environment installed, regardless of operating system.[8] Even if you have never developed applications using Java, there is a very good chance that you've used an application that was written in Java: over 1.1 billion desktop machines run Java and 3 billion phones run Java. Popular entertainment format Blu-Ray players all run Java and applications from games to car navigation systems all run Java.[1]

As well as being compatible with a huge range of systems, Java also has several benefits for developers. Java is very secure as the language, compiler interpreter and runtime environment were all developed with security in mind.[5] Java's Object Oriented nature means that coding in Java revolves around creating Objects which can then be reused and intertwined. It makes complex applications easier to develop. Java also has automatic garbage collection: if an instance of a class is no longer being used, Java will automatically remove it from memory so that the space can be allocated to something else without any input needed from the user or developer.

There are several fully featured Java Development Environments such as Eclipse and NetBeans that make learning Java and Java development very easy. The language itself has a deep repository of guides and information, both official and unofficial, and often features more descriptive and helpful errors compared to languages such as C or C++. The Eclipse and NetBeans JDEs also provide numerous features that make tasks far simpler for programmers, helping to maintain encapsulation in Java applications. For example, if a user created a private variable (a variable that can only be accessed from within the same class as it is defined), but needs to change it from outside the class, the developer needs to create "getter and setter" classes to control access to the variable. Eclipse can create these methods automatically for developers with just a few clicks, saving a lot of time. Eclipse also warns users of errors in their code before they have compiled it, meaning errors can be dealt with as a user writes code, rather than needing extensive debugging every time the user compiles.

Eclipse also allows for user extensions to its functionality. For example, WindowBuilder allows users to develop fully functional Graphical User Interfaces easily and quickly using a GUI interface, rather than having to code interfaces themselves.[2]

Java does has a few disadvantages, however. Java can be significantly slower or memory consuming than natively compiled languages[5], and by default Java GUI applications can look vary different to a native Windows application due to button styles, though these can be customised by a user.

My existing familiarity with the Java language, and the wealth of existing

documentation makes Java an easy choice for building my game. The popularity and prevalence of the Java Virtual Machine means almost anyone would be able to run an application developed for it, and there are a wide variety of tools and resources to help developers including an established development community that can answer almost any question a programmer has.

2.2 The Pathfinding Problem

As mentioned previously, Pathfinding is a major problem that must be solved in order for the game to work correctly. As limitations will be in place, stopping units from passing over other units, or from moving more than a certain number of squares, we need to be able to move units from point to point without breaking any of the rules. If a unit has a movement range of 7 squares and we try to move 8 squares, the game needs to recognise that this is not a valid move. If a player tries to move to another side of a wall, the game needs to find a path around the wall without going over the move limit. As mentioned by Patel[6], pathfinding is planned, rather than reactive. A good pathfinding algorithm will look ahead to find the shortest possible path, rather than simply attempting to travel in a straight line and adjusting as necessary.

Pathfinding algorithms have their routes in Dijkstra's Algorithm, which finds the shortest path from a point in a graph to a destination. With Dijkstra's algorithm, one can find the shortest path with the lowest cost from a given point to all points in a graph at the same time[4]. The algorithm has many practical applications: if vertices on the graph represented cities and edges represented distances between cities, the algorithm could be used to find the shortest route between one city and all other cities, and is also used in network routing protocols.

The basic algorithm for Dijkstra is described in **Algorithm 1**. On a graph $G = (V, E)$ where V is a set of vertices and E is a set of edges, we keep a set of vertices S whose shortest paths from the source have already been determined and $V - S$, the remaining vertices. We also have d, an array of best estimates of the length of the shortest path to each vertex, and pi, an array of predecessors (an array of indeces for each vertex that contains the index of its predecessor in a path through the graph) for each vertex.

Relaxing vertices refers to the process of updating the costs of all the vertices, v, connected to a vertex, u, if we can improve the best estimate of the shortest path to v by including (u, v) in the path to v. This checks to see whether the current best estimate of the shortest distance to $v(d[v])$ can be improved by going through u (making u the predecessor of v).[4]

Faster implementations of pathfinding can be created through the use of heuristics ("rules of thumb"). The Greedy Best First Search algorithm chooses the vertex closest to the goal first. While this method does not always find the shortest path, it does often find a path quicker: if the goal is to the south of a start point, the algorithm will focus on paths that lead south[6]. However, as best first search is a greedy algorithm, it will try to move toward the goal even if it is not the right path, meaning that if it encounters obstacles, it will keep

Algorithm 1 Dijkstra's Algorithm

1. For each vertex initialise d to infinity (initial estimates are all infinity) and pi to nil as there are no connections
2. Set S to empty
3. While there are still vertices in $V - S$:
   (a) Sort the vertices in $V - S$ according to the current best estimate of their distance from the source
   (b) Add u, the closest vertex in $V - S$, to S
   (c) Relax all the vertices still in $V - S$ connected to u

going, even if a path is extremely long.

The A* (A star) algorithm combines the best of both these methods and is one of the most popular pathfinding algorithms and is used frequently in videogames [9]. In order for the A* algorithm to work, the map being searched must be appropriately organised into nodes, much like Dijkstra's algorithm and the Best First Search. In our case this will be our grid, but in a 3D environment these could be specific waypoints or navigation markers. For each node we have the cost of getting from the start node to the current node (that is, the node we have reached in the search) g, an estimated cost from the current node to the goal node h (based on an admissible heuristic, one that will never overestimate the actual cost), and the sum of g and h to create a best estimate f of the cost of the path going through the current node. A lower g value is a more efficient path to the current node. We also have two lists, an open list and a closed list. The former contains all the nodes that have not yet been examined by the algorithm, and the closed list contains the nodes that have been fully explored (every node linked to it has been looked at). **Algorithm 2** outlines the process of the A* algorithm.

The A* algorithm has numerous advantages. Unlike the standard heuristic based algorithm, it will always find the shortest possible path for an admissible heuristic, and unlike Dijkstra's algorithm, it will always move toward the general direction of the goal as long as the estimated cost is lower.

However, A* also has limitations which must be considered. A* is CPU intensive, and if there are a large number of nodes to search through it can bog down a machine significantly, causing poor performance. The performance hit is exacerbated if multiple searches are running concurrently, and can cause games to freeze until an optimal path is found. Another issue is dynamic objects: if a a path has been calculated and something later blocks it, the path would still exist without accounting for the new blockage. As the game will be turn based, this should not be a problem.

Another Pathfinding solution is the Floyd-Warshal algorithm. This algo-

Algorithm 2 The A* Algorithm

1. Let P = starting point
2. Assign f and g values to P (g is the estimated cost from the current node to the goal, f is a best estimate of the cost of the path going through the current node)
3. Add P to the Open list
4. Let B = the node from the Open list with the lowest f value
   (a) If B is the goal node, quit as a path has been found
   (b) If the Open list is empty, quit as a path cannot be found
   (c) Let C = a valid node connected to B
      i. Assign f and g values to C
      ii. Check if C is on the Open or Closed list
         A. If it is, check if the path has a lower f value and update the path if it does
         B. Else, add C to the Open list
      iii. Repeat step 5 for all valid children of B
5. Repeat from 4

rithm simply finds the shortest paths between any pair of nodes. With a sufficiently small grid, we could simply run the algorithm for every combination and then store the results in a table, and then look up the shortest path for a given move whenever it is needed. However, we also need to consider that certain obstacles to movement will be dynamic: if all obstacles were always in the same place this could be an ideal solution. However, since players can block paths, and players will move every turn, the table would have to be recalculated every single turn, which would make it far less efficient.

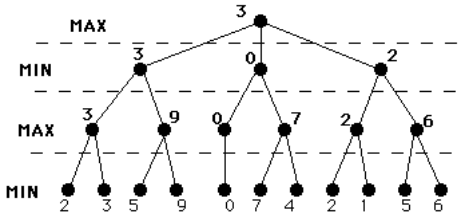
2.3 AI and Minimax

Many operating systems today come with a small selection of games to serve as entertainment. A few of these involve some form of AI, such as Windows Vista and Windows 7's "Chess Titans". There are a few existing algorithms that we can investigate and adapt for the AI player in this game. A key, and very popular approach is the Minimax (occasionally Min-Max) algorithm, which is often applied in two player games from noughts and crosses through to chess. A common element with games that use the minimax algorithm is that they have a tightly defined set of rules, and because of these rules, we can compute what the possible moves are at any given point in the game (in most cases. Some

game's have such a large potential set of permutations that minimax is simply not useful). In addition to this, each player can see the full state of the game: they can see where both their and their opponent's pieces are located and what moves they can make[7].

Minimax is implemented using a search tree. With a search tree we start at the root, then at each decision we create a node for each of the possible moves (similarly to A*), and we keep generating possible moves until a specified depth is reached or no more moves can be generated. With Minimax, we assume the existence of two players: one Max and one Min. The Max player is attempting to maxise some form of score, and the Min player is attempting to minimise it. To use a noughts and crosses example, if the Max player is playing as crosses, a win for X could be +1, and a win for O as -1. Max would be trying to maximise the final score, and Min would be trying to Minimise it[7].

We generate a search tree up to the game's end position and from there evaluates which possible outcome is best for the computer's opponent (in a game like noughts and crosses we simply have win, lose or draw, or +1, -1 and 0. For games where an actual score is trying to be maximised, we can use the actual score of the player at that point or a heuristic evalutation of the strength of the position). The algorithm assumes that if we reach that stage in the game, the Min player will make the move which leads to the outcome that is best for it, and thus we can "know" what the opponent will do if we reach that stage, assuming that they play optimally. The computer can treat the value at the position of that node as a terminal state (one that ends the game, even if it is not), and then evaluate a level higher until we reach the top of the tree. Each option the computer has available can be assigned a value as though it were a terminal state, and then simply picks the highest value and takes the action connected to it[10].



Minimax of a hypothetical search space. Leaf nodes show heuristic values.

In the image above, the computer makes a decision at the levels labelled MAX, and the player at the levels labelled MIN. Each leaf node has been given a value according to a heuristic evaluation, which is then propogated upwards: if the parent level is MAX, we take the max value of the children. If the parent level is MIN, we give it the MIN of its children.

When we reach the top we are at the current turn, and the computer will know the possible outcomes of choosing the move at each node, and then simply choose the move that gives it the highest, or best score. For example, the

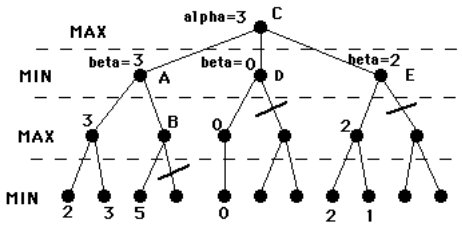
leftmost subtree is the best move because when the opponent makes the move that minimises MAX's score, the best move following this is the higher than the comparable situation in the other subtrees[3].

Optimisation

A way to reduce the amount of computation needed during minimax is Alpha-Beta pruning, which "prunes" away search branches that cannot possibly influence the final decision of the algorithm. Proceeding in a depth first fashion (starting at the root and proceeding as far along a branch as possible before backtracking), an initial alpha value is associated with a MAX node. As a MAX node is given the maximum value among its children, it can only ever increase or remain the same. An initial beta value is associated with a MIN node: as MIN nodes are the minimum of their children, it can only ever decrease or remain the same.

Alpha pruning occurs when we take the alpha value of a MAX node and travel down the tree, if we encounter a MIN node with a beta less than the alpha we won't need to search any lower than that node since the value will only ever decrease or remain at that value and will not be propagated up to its parent. For example if a MAX node's alpha is 5, and one of the descendant's beta is 3, we don't need to search it.

The reverse is also true: if we encounter a MIN node with a beta of 5, and find an alpha with a value of 5 or more, we can stop the search there since the value will never go below 5 and will not be propogated up the tree. This is Beta pruning[3].



Using the example tree above, we would descend to the bottom of the left most branch and calculate the heuristic value of the two leaf nodes, and propogate the MAX to the parent (3). This value is then given to the grandparent (A) as its beta. Since the beta is 3, we know that it will never go any larger. When we descend to A's other grandchildren, we terminate the search if we encounter a value greater than A's beta. The tree is beta pruned at node B since its value will never go lower than 5.

With no more values to search, we know that A's value will be 3, so we use that as its parent's (C) alpha value. C will never be any smaller than 3. We travel depth first to C's grandchildren. When we reach D, the tree is alpha

pruned since no matter what happens on its right branch, it will not be greater than 0.

E is alpha pruned because with a beta value of 2, no matter what happens on its right branch, it cannot have a value greater than 2, meaning that C is 3[3].

A further optimisation would be to set a maximum search depth to limit the maximum number of turns the AI would think ahead. This could even be used to the game's advantage: by changing how far ahead the computer thinks, we can change how good the AI is at the game, as a player that thinks 3 turns ahead should be weaker than one who thinks 10 ahead.

Weaknesses

While minimax can form the foundation of an AI opponent, it also has weaknesses. For example, in a game like chess, mapping an entire game from current point to end could result in an extremely large tree. A similar situation could be encountered with our game: in matches featuring a large number of units and a large map, the number of possible moves could be astronomical and take a very long time to compute. A potential solution to this could be to limit the maximum depth of a tree, or to use heuristic scores to cut off of the tree and estimate how desirable a state is.

In a game where there is no opening move that guarantees victory, if both players were to always play optimally and even if the computer could calculate every possible situation (which would take enourmous amounts of computing power), the best the computer could ever do would be to draw (which could be true, if both players had absolutely perfect knowledge of every possible move)[10]. However when combined with heuristics, a minimax algorithm can calculate probabilities that a move will lead to advantageous situations. In chess pieces are often assigned values that evaluate how useful their are for a game victory: if a pawn were scored at 1 and a Queen at 9, the algorithm would conisder sacrificing pawns as a way of maximising its score by eliminating the opposing Queen.[10]

3 Approach

This section covers how I have chosen to achieve the aims and objectives covered in the introduction.

3.1 The Grid Implementation

The grid for the game will be implemented in Java using a pair of two-dimensional arrays. The first of the two arrays will store the actual map that players are playing on: location [3][2] in the grid might contain water, or [14][2] may be grass, for example. The second array will contain unit positions: for example

[4][5] in the grid may contain a red unit, and [6][15] a red unit. This implementation will be convenient because it separates out dynamic and non dynamic elements of the game: units will constantly be changing position on the map, but the map itself will not change over the course of the game. Storing the two in the same grid could be confusing, especially when trying to pass information to a pathfinder. With this implementation, we should simply be able to check one array if a unit is where the player clicked, then evaluate the terrain on the other array to build a path to where the unit wants to move.

3.1.1 Grid Cell Types

The plan is for a small number of cell types: a standard floor tile that can be moved and fired over, a wall tile that can neither be fired or moved over, and a water tile that cannot be moved over, but can be fired through. It is likely we could represent these in the grid array without dedicated classes: for example, if a wall was represented by 1, water by 2 and the pathfinder evaluated a cell with either of these values in, it could have a simple class to check if a cell is blocked consisting of a simple if statement. The only remaining cell type would allow movement and firing through, so could simply the default case. With walls being the only type that cannot be fired through, their locations could simply be stored in an additional array that can be checked by the shot legality algorithm as required.

To visually distinguish grid cells, we can simply use different colours: the ground could be a green colour, signifying grass. Water could be blue, and walls grey. This would instantly signify what a cell in the map is to a player. When it comes to drawing the cells, Java's graphics package can simply iterate over the whole grid and draw a specific image file inside the game window based on what value is contained in the cell of an array. The graphics package could use the index of the array to work out where to draw: if the image tiles used are 64 by 64 pixels and we find a value representing water at [3][3] in the array, we could simply multiply each co-ordinate by 64 to find out the exact pixel co-ordinate in the window to draw the image.

3.1.2 Map Generation

Whilst preset map layouts will be selectable, players will also want to generate their own layouts. If I implement the integer based terrain grid described above, I could easily use a GUI interface to design maps: clicking on a cell will simply iterate the relevant cell in the array from 0, to 1, to 2 and then back to 0. This would also make saving map layouts easy: the grid of integers and grid of unit locations could simply be exported and imported from text files, with the map values set according to those files and unit instances generated as necessary.

3.1.3 Move Legality

Even with the problem of player movement covered by pathfinding, other issues involving move legality need to be considered. Whilst the concept of the game

may be similar in practice to a board game, the mechanics behind attacking other units requires serious thought. When a unit attacks another, there are two checks that must be passed. The first a simple check that the unit is within range: if there are more than n squares between unit A and unit B, then the unit B is out of range. If unit B is in range, we need to check if unit A can 'see' unit B, which is an altogether more difficult problem to solve.

If units were limited to shooting in straight lines (pure horizontal and vertical lines, or pure diagonals consisting of an increase/decrease of 1 in both x and y co-ordinates), checking if an object is in the way would be a simple task as we could simply iterate through every cell between the two points and check if it is a wall, or to save computing time, store wall locations in an array and check if any of their co-ordinates crosses the line. However this approach does not appeal: since I am not restricting units to purely horizontal or diagonal moves, it would seem incongruous to enforce such an arbitrary limitation on attacks. An additional worry of such an approach would be never-ending battles where one player constantly tries to move into a position where they can attack whilst the other constantly moves away due to needing a very specific angle to attack. If there were no wall tiles in the game, a visualisation of a units possible moves would appear like a diamond shape (or a square if diagonal moves were allowed). If a unit can move anywhere within such a radius, it makes sense that they would be able to shoot somewhere within a similar one.

To achieve this, I need to check if any points along the line between two points pass over a region that would block a shot. To do this I will need to draw a line between the two units and check to see if the line that is created passes over any wall cells. The basic approach is to treat the game grid as co-ordinates on a graph and plug those co-ordinates into the line equation created between the units and comparing the resultant values with the values of the wall tiles. **Algorithm 3** covers this in detail.

Worked example

If unit A is in cell $(2, 2)$ and unit B is in cell $(4, 6)$, we get the line equation $y = 2x - 2$. If we now take a wall cell situated in $(2, 3)$, we can plug x co-ordinate of the wall into the equation to get $y = 2$. However, as the graphical representation of the cell will stretch from 2 on a grid to the start of 3, we also need to test for $x = 3$. This gives us $y = 4$. As a result we have a range of y values of 2 to 4. If we compare this range to the possible range of y values our wall cell can inhabit (the cell is at $(2, 3)$ so can stretch from $y = 3$ to $y = 4$), the ranges overlap. As they overlap, this means that the cell blocks the shot

If a wall is in cell $(3, 2)$, using x values of 3 to 4 results in a y range of 4 to 6. The wall will stretch from 2 to 3, so these two ranges do not overlap, and so cell $(3, 2)$ does not block the shot.

Algorithm 3 Shot Legality Checking

1. Use the co-ordinates from the unit positions $(x_1, y_1), (x_2, y_2)$ to calculate the change in y ($\triangle y = y_2 - y_1$) and and change in x ($\triangle x = x_2 - x_1$)
2. Do ($\triangle y \div \triangle x$) to calculate m
3. Use co-ordinates of the shooting unit ($x_1 y_1$) to complete the line equation in the format $y = mx + c$
4. For each item in the array holding the co-ordinates of wall tiles
   (a) Use the wall's (x_3, y_3) x value as x in the line equation to calculate a y value
   (b) If this y value is between the y co-ordinate of the wall cell (x_3, y_3) and $y_3 + 1$
      i. The shot is blocked
   (c) Else, the cell being tested does not block the shot

3.2 Multiple Unit Types

The units will need to be both distinctive in function, but also behave in a similar way. For example, the bishop and rook in chess are identical, except one moves diagonally, and the other in straight lines. In a programming representation, they would likely use similar class definitions as any other piece in the game, simply containing a subclass defining how a piece is allowed to move. The same will be true of my game: whilst pieces will be able to move different distances per turn and shoot over different distances, their underlying implementation needs to be very similar to ensure each piece is compatible with the pathfinder and game logic.

To achieve this we can use a Java interface to define a Unit Interface that every unit class must inherit from. This interface could contain methods for retrieving and setting the unit's position, retrieving its maximum move or attack distance, and so on. This would ensure consitency across the classes.

This would allow us to implement the grid of unit positions as a custom data structure similar to how we might have a grid of integers to represent the game map. Each cell would have an x co-ordinate and y co-ordinate along with a reference to a unit instance that can be set to anything that inherits the correct interface. If nothing is in the cell, it could be cet to null. This would allow units instances to exist seperately from the implementation of the two grids ensuring encapsulation and allowing us to easily track each unit's status.

3.3 Multiplayer Functionality

Multiplayer will be achieved through a simple turn based structure that switches which pieces can be moved after after turn. A value could be set to either 0 or 1: if it is 0 it is player 1's turn, and if it is 1 it is player 2's turn. Once a player has moved, fired or ended their turn manually, the value would switch. Each unit could have a similar value identifying its team: when the turn ID is 0, units with a team ID of 0 will be allowed to move, and vice versa. This value could also be used to ensure that a unit is attacking units on the other team, rather than itself or units on the same team.

Simple variables could be used to track turn counts, how many victories each team has achieved, and so on.

3.4 Pathfinding

Pathfinding will be implemented using the A* method as explored in the previous section. The A* pathfinding algorithm allows excellent accuracy, and since units in our game will have limited move distances, searches will never reach a depth that could slow a modern machine down, and only a single search will ever be run at once, since only a single player can move a single unit at any given time. It is also relatively simple to implement as the core algorithm is simple to understand.

3.5 Artificial Intelligence

The game will employ a minimax and heuristic based AI. Using this type of AI will allow the player to customise how the AI behaves: the number of turns the AI thinks ahead could be changed to provide a more or less challenging opponent, whilst different heuristics could be employed to prioritise different situations, providing the AI with different personalities. Heuristics could even be changed mid game based on the situation (if the AI is losing, switch to a heuristic that encourages more defensive play, for example). A probability based chance for the AI to deliberately choose a bad move could also be implemented: on high difficulty settings, the AI will think a large number of turns ahead and will almost always choose the best move. On low difficulty settings the AI will think a small number of turns ahead and has a larger chance to make a poor move. By choosing an AI implementation that allows customisation, the game should be enjoyable for players of any skill level.

3.6 Further Features

Units will be designed as small 2D .png images using a program such as Adobe Photoshop or Paint.NET. 2D sprites can be easily distinctive, and until polygon based visuals arrived in the mainstream, were extremely common in video games. 2D sprites will also cut down on processing power needed to render the game. To make units visually distinctive they will be color coded by team, and

each type of unit will have its own design and be shaded slightly differently to units on the same team.

4 Conclusions

In this interim report I have outlined the key aims and objectives for this project, as well as my research into solving the problems that these objectives have introduced. I have chosen a suitable programming language with which to develop the game, and outlined algorithms for solving the problems of pathfinding and providing a challenging, customisable AI opponent. I have also explained how I plan to approach implementing the project, including a solution for computing whether attacks are legal using line calculations. If I follow the approach outlined and implement my background research correctly, I should have no problem developing a strong final product.

References

[1] Learn about java technology. Website. Available Online: http://www.java.com/en/about/.

[2] Google. Windowbuilder user guide. Available Online: https://developers.google.com/java-dev-tools/wbpro/.

[3] Ralpha Morelli. Cpsc 352 – artificial intelligence notes: Minimax and alpha beta pruning. Website. Available Online: http://www.cs.trincoll.edu/ ram/cpsc352/notes/minimax.html.

[4] John Morris. Dijkstra's algorithm. Website, 1998. Available Online: http://www.cs.auckland.ac.nz/ jmor159/PLDS210/dijkstra.html.

[5] MS. Java advantages and disadvantages. Website, April 22nd 2007. Available Online: http://www.webdotdev.com/nvd/content/view/1042/204/.

[6] Amit Patel. A star comparison. Web Blog. Available online: http://theory.stanford.edu/ amitp/GameProgramming/AStarComparison.html.

[7] Paulo Pinto. Minimax explained. Website, July 27 2002. Available online: http://ai-depot.com/articles/minimax-explained/.

[8] Jason W. Purdy. Write once, run anywhere? Website, February 9th 1998. Available online: http://www.developer.com/java/other/article.php/601861/Write-once-run-anywhere.htm.

[9] Hugh McCabe Ross Graham and Stephen Sheridan. Pathfinding in computer games. School of Informatics and Engineering, Institute of Technology, Banchardstown Available Online: http://gamesitb.com/pathgraham.pdf.

[10] Stanford University. Algorithms - minimax, strategies and tactics for intelligent search. Available Online: http://www.stanford.edu/ msirota/soco/minimax.html.