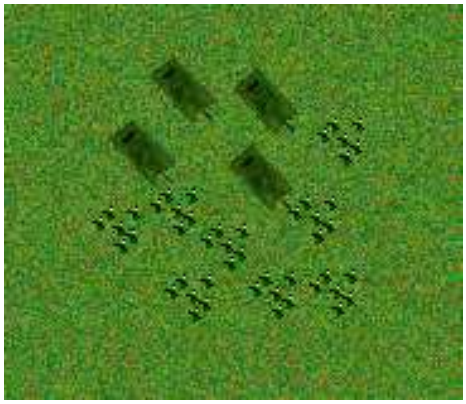
# JWars - A Generic Strategy Game in Java

Midterm Project at IMM



---

**Michael Francker Christensen s031756**
**Ask Hjorth Larsen s021864**

Supervisor: Paul Fischer

DTU - Technical University of Denmark

June 27, 2006

*Front page: Soviet T-34 tanks supported by infantry advancing across the Russian steppes*

**Abstract**

# Contents

# Chapter 1

# Introduction

## 1.1 Acknowledgement

## 1.2 About the real-time strategy genre

When categorizing Jwars it should be specified as a Real-Time Tactical game. This genre however belongs under the broader type of games called Real-Time Strategy ( RTS ) which is normally used. The RTS genre came about in the 80's only being fully developed and formally seen as a single genre 10 years later with titles such as *Dune II* and later Blizzard's titles *Warcraft* and *Warcraft II*. For the casual gamer a RTS game can be recognized by using some simple ground rules which have grown to distinct the genre:

- Warplanning is essential - strategy
- The player has no 'Next turn' button - real-time

Other essential guidelines:

Resource gathering Building/unit locations are essential The manufactoring of specific units The player has direct control of all his units/buildings

The RTS genre was developed from the turn-based strategy games genre. One of the first RTS games, perhaps the most important one, is dune II for which the developers was inspired by Sid Meiers Sim City. It should be noted that while Sim City differs from the standard RTS game, it is also recognized as a RTS game where the opponent is the game environment itself and not an AI or another human player.

RTS games today is normally played 'player vs player' either online or on LAN connections. Most RTS games developed recently has focused mainly on the gameplay in multiplayer form than the gameplay in single player form with mission.

## 1.3 Why JWars?

# Chapter 2

# Features of JWars

## 2.1 Game dynamics

game design regarding hierarchy

## 2.2 Technical features

This section lists briefly the

- World representation. JWars uses a number of abstract 2D coordinate spaces and provides utilities for conversions between these. Specifically many *tile*-based maps are required by the different components of JWars.
- Collision detection. An efficient tile-based collision detector is capable of detecting collisions between circular objects of arbitrary size.
- Pathfinding. The pathfinder implements an A* algorithm which dynamically expands the search area according to requirements. This approach accomodates obstacles of arbitrary size and placement.
- Spotting system. The spotting system uses a tile-based approach which is particularly efficient if the map is large compared to the visibility radius.
- Artificial intelligence. A simple but highly extensible
- Event handling model. A queueing system provides efficient management of timed execution of game events avoiding unnecessary countdown timers.
- Data management. Script-like files can be used to store game data such as unit and weapon statistics. These are loaded into *categories* which represent the abstract concepts of those units or weapons. Finally entities can - in turn - be instantiated from categories.

- Server-client based networking model. The TCP/IP based networking model supports a customizable set of instructions and provides base server and client classes for managing player connections. This model has very low bandwidth requirements, but requires perfect synchronization of the game states across the network.
- Multiplayer synchronization utilities. Synchronization on multiple clients is done by means of a timer which assures that clients follow the server temporally closely.

# Chapter 3

# Overview

For reasons of extensibility, JWars consists of several modules which can be used separately or with a minimum of cross-package dependencies.

## 3.1 Development plan

## 3.2 Modular overview

Describe basic concepts such as units

# Chapter 4

# Networking

While real-time strategy games traditionally include single-player campaigns, experiece shows that the success of a game is largely determined by its playability in multiplayer. The online playability of a real-time strategy game is therefore *very* important, and the networking implementation can have profound impact on this[1]. This chapter will explore the options available to JWars and in turn decide on a feasible design.

## 4.1 Choosing a network model

There are several different architectures and protocols used in multiplayer games, and different genres have different requirements regarding efficiency and response times. Fundamentally we shall discuss two variables in this entire problem. First there is the amount of game data which has to be synchronized across the network, along with the and the response time, i.e. the *ping* or *latency*.

We can roughly categorize real-time computer games by their networking requirements:

1. Small, fast-paced games such as first-person shooters. These games require low ping but have small amounts of data to synchronize (e.g. the positions and speeds of a few dozen game objects). For example the game *Counter-Strike* is usually played by around 10-20 people who each controls *one* person, and network latency can quickly cause deaths in the fast-paced firefights.

2. Large, slow-paced games such as real-time strategy games. There are very large amounts of data (hundreds or thousands of game objects), but there are only lax requirements to response times since the player is not concerned with such low-level control as above.

[1] *Command & Conquer: Generals* is regarded by the authors of this text as one of the finest real-time strategy games ever conceived, and yet this game remains largely unplayed online. Even on a high-speed LAN the game speed will almost grind to a halt with just four players. Our conclusion: *they chose the wrong network implementation.*

3. Large, fast-paced games such as massively multiplayer online role-playing games. These require both fast response and involve very large amounts of data, and therefore demand very advanced networking code. It is well known that this takes its toll even on modern games of the genre, but luckily this is none of our concern.

We are obviously concerned only with the second category. We note two ways to keep the game state identical across a network: either we can beam the *entire* game state consisting of *every* logically significant game object across the network with regular intervals. This approach obviously only accomodates games of the first category because of sheer bandwidth requirements. Another – and to us better – way is to let *every* computer simulate the *entire* game logic in parallel, and only send across the network those instructions that are issued by the *players*.

This approach is promising since it requires next to no bandwidth even though thousands of units are on the battlefield. However it is strictly required that all comptuers on the network are able to perform exactly the same simulation given the player inputs received from the network, otherwise the game will go 'out of synch' and never recover. The next section will describe this approach in detail.

## 4.2 Synchronization

We shall now propose a complete solution to managing the flow of time (in the game, that is). Suppose until further notice that the players have no control of the game. We define that the game starts at *frame* 0, or $t = 0$, in some initial *state* which is identical on all those computers that partake in the game. Now, all the partaking computers will perform a *logical update* (which will allow entities to move or fire at each other automatically and *deterministically*, i.e. without the player issuing instructions) at regular (and equal across the network) intervals, and when such a logic update on some computer is completed we say that the frame count $t$ is increased by one on that computer. Thus, as time progresses every computer will execute further logic updates for $t = 1, 2, 3 \ldots$ until the game is over, and if the logic update routine is *consistent* then the computers will all be in the *same state at all time*.

There is no network activity yet since the logic update routine is deterministic and therefore requires only local information. Note that the computers do not need to execute the same logic update at exactly the same *physical* time, the only important thing is the relationship between frame count and game state.

#### Interactivity: player instructions

Suppose now that we will allow a player to affect the game state, which is hardly a deterministic endeavour (except in Chartres' philosophy; however we shall here define deterministic as something which a computer can predict, seeing as the deeper philosophical considerations go beyond the scope of this text).

We will need to send the particular instruction that this player has issued to *all* computers in the game such that they can execute it. Furthermore it is obviously vital that all computers execute this instruction while in the *same frame*, otherwise they will go out of synch forever.

Let us say that some computer acts as a *server* which keeps track of the frame count, while all players are *clients* connected to the server. The player who wishes to execute an instruction then sends that instruction to the server. The server receives this instruction while in frame number $t_0$. Now, every computer on the network must receive this instruction and execute it at the same time, so the server echoes the instruction to all clients *along with* the requirement that the instruction be executed at frame number $t_0 + L$, assuming that the instruction will arrive to the other computers before they have furhter executed $L$ updates (we shall refer to $L$ as the *latency*, even though the physical ping results in a slightly larger actual latency). Now, each client will receive the instruction and can enqueue it for execution in the $(t_0 + L)$'th logic update.

### Synchronization instructions

What happens if the instructions arrives late to one player, at time $t_0 + L + \delta$? Then that computer will no longer be able to execute the instruction in time, and the game is ruined forever. This must not happen, and we shall therefore require that the server provides as a *guarantee* to each client that they are allowed to execute updates until some frame count. If the server continously sends out *synch* instructions to all clients stating that they may proceed the updating procedure until frame $t$ where $t \leqslant t_0 + L$, then a client can halt the game flow if it reaches time $t$ and not continue until receiving a new such instruction from the server. In the meantime any instructions that arrive will be enqueued for execution at times later than $t$, ensuring their eventual execution at the correct time.

A game implementing the ideas presented here will not rely on a classical game loop which performs updates at the highest possible speed, but instead use a timer which updates only at regular intervals. It is still possible to render at higher frequency than the logical update rate, using interpolation, see section **??**.

### Conclusion

We now have a completely synchronized model which supports any number interacting players and requires a server. The network activity will be very low, perhaps few instructions per second for synchronization and a term proportional to the player activity. Since the server will have to send each instruction to $n$ players, and $n$ players will send $\mathcal{O}(n)$ instructions, the bandwidth use will be $\mathcal{O}(n^2)$ unless special countermeasures are taken, but real-time strategy games are traditionally played by no more than around 12 players, and with the low per-player bandwidth requirement this remains acceptable.

## 4.3 The networking API

The objective of this section is to design a networking package adhering to the requirements specified in the previous section. This will be done in an *event-driven* way which exposes a continually updated instruction queue to the programmer who can therefore integrate it asynchronously in any timer based or game-loop based implementation.

The instructions considered in the previous sections, both synch instructions and client instructions, obviously require guaranteed delivery in consistent order. Both of these properties are ensured by the TCP/IP protocol, and along with the lax latency requirements this shows beyond doubt that TCP/IP is better than UDP for our purposes.

It is evident that we need a `Server` and a `Client` concept, along with the concept of *instructions*. We shall further introduce the *protocol* which is simply a collection of instructions.

The entire client-side networking model consists of a

# Chapter 5

# Event handling

# Chapter 6

# World of JWars

## 6.1 Coordinate systems

## 6.2 Data management

## 6.3 Terrain

# Chapter 7

# Collision detection

This chapter will after an introduction to collision detection describe the design and capabilities of the JWars collision detector.

## 7.1 Introduction to collision detection

The most important objective of this section is to decide on an overall approach to an efficient and reasonably simple collision detector bearing in mind the requirents of real-time strategy games. There is by no means an *optimal* such collision detector since requirements invariably will differ greatly with applications. Further shall restrict the discussion to two-dimensional collision detection seeing as JWars does not need three dimensions.

In a real-time strategy game there is generally a large amount of units, possible more than a thousand. It is therefore of the utmost importance that the collision detector *scales* well with the number of units in the game.

Let $n$ be the number of units present in some environment. In order to check whether some of these overlap it is possible to check for *each* unit whether this unit overlaps *any* of the other units, and we will assume the existence of some arbitrary *checking* routine which can perform such a unit-to-unit comparison to see whether they collide. While the amount of such checks can easily be reduced, for example noting that the check of unit $i$ against unit $j$ will produce the same result as the check of unit $j$ against unit $i$, this method invariably results in $\mathcal{O}(n^2)$ checks being performed. This approach is fine if there are very few units, but this is obviously

The amount of checks can, however, be reduced by *registering* units in limited subdomains of the world and only checking units in the same subdomain aganst each other (for now assuming that units in different subdomains cannot intersect). Suppose, for example, that the world is split into $q$ parts each containing $\frac{n}{q}$ units. Then the total amount of checks, being before $n^2$, will be

only

$$\text{number of checks} \approx q\left(\frac{n}{q}\right)^2 = n^2/q.$$

It is evident that within each subdomain the complexity is still $\mathcal{O}(n^2)$, but decreasing the size of the subdomains can easily eliminate *by far* the most checks, particularly if the division is made so small that only few units can physically fit into the domains. The applied approach thus employs principles of a *divide-and-conquer* method[2, pp. 28-33], though it is not explicitly recursive.

This approach still needs some modifications in order to work. Specifically, units may conceivably overlap multiple subdomains, necessitating checks of units against other units in nearby subdomains. Assuming square subdomains will prove both easy and efficient, and we shall therefore do so. Consider a *grid* consisting of $w \times h$ elements, or *tiles*, defining these subdomains–see figure **??**. We shall describe two ways to proceed.

1. *Single-tile registration.* Register each unit in the tile $T$ which contains its somehow-defined geometrical center. In order to check one unit it is necessary to perform checks against *every* unit registered in either $T$ or one of the adjacent tiles. Thus every unit must be checked against the contents of *nine* tiles. This approach is simple because a unit only has to be registered in *one* tile, yet much less efficient than the optimistic case above and requires that the units span no more than one tile size (in which case they could overlap units in tiles even farther away).

2. *Multiple-tile registration.* Register the unit in *every* tile which it touches (in practice, every tile which its *bounding box* overlaps). Checking a unit now involves checking it against *every* other unit registered in *any* one of those tiles it touches. This means that a unit whose bounding box is no larger than a tile can intersect a maximum of four tiles. Units of *arbitrary size* can cover any amount of tiles and therefore degrade performance, but the collision detection will obviously not fail–also in most real-time games the units are of approximately equal size and for the vast majority this approach will be .

For the JWars collision detector we have chosen the second approach, primarily because it does not restrict unit size to any particular scale. This approach will also likely be more efficient since it in most cases will require less than half the number of tiles to be visited (as noted, 4 is a bad case in this model whereas the former model consistently requires 9). However there is one possible problem which is illustrated in figure **??**, namely that two units which occupy two of the same tiles will (unless carefully optimized out) be checked against each other in each of those tiles[1].

[1]The present implementation does not optimize this, since this can hardly degrade efficiency considerably.

The best-case time of such a tiled collision detector is $\mathcal{O}(n)$ corresponding to the case where all units are in separate tiles. The tiles should be sized such that only a few units (of a size commonly found in the game) can fit into each, but they should not be so small that every unit will invariably be registered in multiple tiles. Every time a unit moves the tiles in which it is registered will have to be updated, which becomes time consuming eventually.

As an example, this model should easily accommodate a battlefield with many tanks (around $6m$ in size) and at the same time provide support for a few warships (around $100 - 300$ metres). If necessary, it is possible to improve the model by allowing variably-sized tiles, such that the tiles are made larger at sea than at land, for example. This approach will, however, not be implemented since such extreme differences in scales are very uncommon in the genre.

Having covered the methods necessary to minimize the number of *checks*, it is time to briefly mention the checking routine itself. It is obvious that a large-scale game can not realistically provide collision detection between arbitrarily complex shapes. In this genre units are commonly modelled as circular or square, and we have therefore decided to provide only collision detection for circular units. However the JWars collision detector does provide an escape mechanism ensuring that units can implement a certain method to provide *any custom-shape* collision detection. Using circular shapes provides the benefit of simplicity and efficiency, and no custom shape handling will be discussed in this text.

## 7.2 Design of the collision detector

The collision detector manages a basic kind of entity which we shall refer to as a *collider*. The most basic properties of a collider are its location $(x, y)$ and the radius $r$ of its bounding circle (it has a few more properties which are irrelevant to this section but will be mentioned later). These properties are used to determine which of the aforementioned tiles the collider overlaps.

In order to represent the collision grid, the collision detector uses the *map* utility package which is described in section **??**. It requires two coordinate systems: a *main* coordinate system (the $x$,$y$ and $r$ properties of colliders are presumed given in this system) and a more coarse *collision grid*. The latter map is a *tile map* consisting of *collision tiles*, where a collision tile is capable of storing a list of colliders.

Registration of a unit in the collision grid uses the coordinates and radius of the collider to derive a bounding box, which is easily compared - through the coordinate transform provided by the `map` package - to the grid elements of the collision map. The checking routine described in the previous section is easily implemented by traversing the tiles thus overlapped by the collider, then and for each tile comparing the radii of present colliders.

The actual checking routine, `check`, takes a collider and a *desired* location $(x, y)$ as parameters and returns whether the specified location is legal (i.e. does not overlap with any other collider registered in the collision grid).

The collision detector further has a `move` method which takes similar arguments, and which will also *move* the specified entity instead of only performing a check.

### Further features

Finally a few utilities of the collision detector should be mentioned. Since it is desirable to use the collision grid in order to localize units to be *rendered*, a collider can also possess a *sprite*, see section **??**. The sprite is used to track movements on the screen, such that redrawing can be skipped in regions where no such movement takes place.

As another feature, some entities may naturally be able to move past another while others are not. For example, infantry units consisting of multiple men would be able to enter a building which would be impassable by larger objects such as vehicles. Also infantry squads would be able to walk through each other, whereas an infantry unit would not be able to move past a tank (which is massive), and two tanks would not be able to drive through each other. Therefore the collider should also specify a boolean which determines whether the object is *massive*. If either of two colliding colliders is massive, then the collision detectors `check` will return false. Thus infantry squads can easily be made to pass through each other or buildings (all non-massive entities).

Finally it is sometimes desirable to "cheat", i.e. not perform strict collision detection in order to make the gameplay smoother. For example if it is desired that a new unit should enter the map, but there is no space at the desired location, it might be best to disable the collision detector and allow that unit to overlap others until such time as the unit no longer overlaps them (when they or the unit have moved). Colliders may therefore be declared as *ghosts*, in which case the collision detector completely ignores them until they are declared non-ghosts.

Regarding implementation, these two properties, whether colliders are massive or ghosts, are conveniently encapsulated in a set of *collision properties* which every collider must have. The collision properties may be retrofitted in later versions to support an abstract notion of *height* (cf. the "2.5 D" geometry, section **??**) or other concepts that can desirably be modified.

The concept of *colliders* is contained programmatically in the interface `Collider`, such that any class can implement it.

## 7.3 Efficiency and optimization

At an update speed of 50 Hz, the present implementation of the JWars game can on the authors' test systems support approximately 1000 simultaneously moving units before lagging behind in logical framerate. It is, however, possible to run a logical framerate of e.g. 10 Hz and perform interpolation to ensure

graphical smoothness between logic updates[2] (thus using a higher graphical than logical update rate). Using such an approach the performance could be enhanced 10-fold. This is not quite necessary in the JWars application. The collision detector therefore supports around $10,000$ moving entities, but this figure can be reduced if custom geometries are used or if other parts of the logic are computationally heavy.

## 7.4 Conclusion

This chapter has introduced the JWars collision detector, and selected a *tile-based* approach to ensure that the detector accomodates large amounts of entities efficiently.

It works by registering entities in appropriate tiles using axially aligned bounding boxes. Collision checks are done using the *radii* of entities, meaning that all units are considered circular. However an escape method is provided that allows arbitrary geomtry.

Performance-wise

[2]Note that if the update rate is further reduced it will most likely become visible to the human player even if graphical interpolation is performed, since the logical framerate governs firing and other things that are directly visible to the player.

# Chapter 8

# Pathfinding

When moving units in the world of JWars a navigational problem arises when trying to find the shortest paths between to points. There exists a range of solutions when finding the shortest path between to points. These solutions however have different requirements for the map in which to navigate and some might be inconsistent in speed.

Most of todays RTS games solve this problem by using a tile system for the map and designating tiles with either 'used' or 'free' as markers when scanning through the map with an algoritm. This approach has several advantages, like high and consistent speed, while it requires a predefined map-structure to search in. A good example is the A* algorithm which is a shortest path graph algorithm. For finding a shortesth path, using graphs for data representation will be the best way. When finding a path on a map you will need fixed points (reference points) designating where to turn and to make heuristic evaluations during the search. A graph is normally represented like this:

$$G = (V, E).$$

// indsæt illustrationer

$V$ is a list or other representation of all the *vertices* in the graph, and $E$ is a representation of the *edges* in the graph. An edge is best seen as a link between two vertices - meaning that you can go from vertex $v1$ to vertex $v2$ using the edge $e(v1, v2)$. The weight of an edge, corresponding to the amount of time it takes to traverse it, is given by a weight function $w : E \mapsto [0, \infty[$ since a distance already travelled can not be negative.

Given a graph with a chosen data structure there are multiple ways to solve the single-source shortest path problem from vertex $A$ to $B$. Most of these algorithms are based on selective expansion of the search area since this type has the best running times with the fewest vertices visited.

The pathfinding in our instance has some requirements to the algorithm which we must take into account before implementing a final algorithm. The most pressing issue is to ensure compatibility with the very general approach

taken in the JWars world representation (for example entities can be very differently sized, and it is assumed that the world might be much larger than in most contemporary real-time strategy games). We have thus chosen a very open approach in the area of unit location and building placement.

It is possible to entirely define the physical constraints of an entity according to *tiles* which it occupies instead of using a continuous system as we did in chapter **??**; however, as we have seen from the collision detector, we use greater precision than a tile based approach offers. This approach introduces one significant problem to a tile-based pathfinder, namely that tiles will frequenty neither be completely *occupied* by entities nor completely *free*. This purely tile-based collision system seems[1] to be implemented in some games and works well - in these games however the placement of objects suffers from the above mentioned limitations.

A tile system is incapable of handling the pathfinding in JWars - the aspect of pathfinding on graphs is however still viable and the most efficient method. The implementation we have chosen for the pathfinding is to transform the dynamic/open implementation of the JWars-world to a graph-system on which we can perform a search algorithm. For accomplishing this we have implemented a dynamic graph system explained in the next chapter.

For every path needing to be found we start with the given graph for the current map $G = (V, E)$. V consist of all corners on static objects convex hulls on the map. This data is stored in the collion map. $E$ starts of as an empty list. [2]

The start and goal location are added to the program as Sloc and Gloc - these are considered vertices which is set for each running of the algoritm. In general pathfinding A* is considered the most effective search algorithm on the single source shortest path problem. In effect this means that the algorithm will terminate as soon as the shortest path has been found to the given goal vertice. There exist a number of algorithms to solve the problem but the A* algorithm has the shortest running time and fits or problem profile well in the almost greedy expansion of the search tree.

In theory all edges can be represented in $E$ - but not active until discovered by the algorithm. Using this approach we expand the graph according to A* and evaluate it as such. The operation that makes this algorithm stand out is the *shoot-to* function which activates vertices/edges while searching for the path. **FIXME**[3]

The important aspect of the chosen solution is that it is not affected by any other part of the game implementation except the collision detector. If a developer wants to use this pathfinder it is fairly easy to convert to a completely different setup - all that is needed is a function that can detect a collision on a line

[1] In *Starcraft*, for example, buildings can only be placed at discrete intervals, and equally sized units tend to line up in *grid*-like formations.

[2] If it were to be a pre-defined list it should consist of all possible routes between any vertices on the map. This amount of data would be hard to handle and if the amount of static objects were large enough it would require alot of memory space.

[3] fixme: So how does it 'activate' vertices/edges?

between two coordinates (which will automatically be traversed by the shoot-to function). This makes the final pathfinding solution as flexible as possible and not letting it have any depencies as collision grids or even a grid based movement system**FIXME**[4]. The pathfinder does, however, have some limitations. The pathfinder will be able to fall short if objects are not given as either circles or convex polygons. **FIXME**[5]This small bug only arises when a player gives a move order inside an objects convex hull. Here is a quick overview of the different methods and the relations described.

```
pre-settings:
all vertices/pathfindingnodes have been initialized with h = infinity
C is the list of vertices to expand - the openlist.


FindPath(Sloc, Gloc)

openlist.add(Sloc);
current = openlist.pop();

do{
if(current = Gloc){
backtrack();
}

shoot(current, Gloc);
openlist.sort();
current = openlist.pop();

}

while(openlist.empty() != false)

clear();
failed();
end

Shoot(From-Node, To-Node)

expand1 = null;
expand2 = null;

collider = walk(From.coord, To.coord)
```

[4]fixme: But how can it not depend on a grid when it has a shoot-to function?
[5]fixme: Why would this be? There is no description of this.

```
if(collider != To){
collider.expand();

if(collider.size != small){
shoot(From, expand1);
shoot(From, expand2);
}
else{
openlist.add(expand1);
openlist.add(expand2);
}

else{
To.update(From);
openlist.add(To);
}
end

Backtrack

S := empty sequence
u := t
while defined previous[u]
      insert u to the beginning of S
      u := previous[u]
end
```

**FIXME**[6] The main methods are quite simple, though not all aspects have been included here. We start the pathfinder by calling the FindPath function with the given start and goal coordinates. The openlist is used for expanding the algorithm and will contain all nodes which are marked for further expansion.

The actual algorithm takes part in the shoot-function which will try to expand the search area by either calling itself recursively or simply adding the found corners to the openlist. When all the relevant object corners have been found, updated and added to the list, the algorithm will sort the list and select a new node for expansion until the goal has been reached. When updating a node we call a method within the PathFindingNode class. This method will update three values which is needed for sorting and evaluating nodes in the list. This is done so the search area can be expanded further according to the heuristic evaluation. Finally the method will set the ancestor of the given node to the node from which we came to make backtracking possible when a path has been found. **FIXME**[7]We use values $f$,$g$ and $h$. $f$ for the travelled distance to this

[6]fixme: Separate interface and implementation. Don't mention openlist (invisible to the user) while describing how the pathfinder is used.

[7]fixme: f,g,h: they denote the values regarding 'this' node, huh? So each node has an

node, $g$ for the heuristic evaluation to the goal and $h$ as a representation for the combined values.

The heuristic evaluation is based on an evaluation in a 2D environment for pathfinding. Taking into account that all distances travelled are straight lines, we can always be sure that we have the shortest possible path to any given node if the 'Relax' **FIXME**[8] method is used. This is described in [**?**] when describing Dijkstra's algorithm. The $g$-score for a node is simply calculated by taking the distance from the current node to the goal location. The g-potential will ensure that a node having travelled less than others and having the possibility to result in getting directly to the node will be the next expanded. This approach mean we can safely terminate the algorithm upon reaching the goal location because the given path will be shortest path possible. Any further expansion of the algorithm will not be neccesary.

The algorithm is not very efficient since it has to double check some values. To somewhat eliminate this problem a minor fix has been implemented. All buildings have a boolean, 'small', attached to them (the default setting is false since it is only a help for a developer). When constructing a building by chosing a *blueprint* the developer can choose to make it a category small object if he see fit. This is meant for creating objects which a unit colliding with should not deviate noticeably from its original course to get around - thus saving the time for checking the routes to the corners of the object in the shoot function. If the object is small enough a shoot function very similar to the the original would be called two extra times. If the small boolean is set true on large objects it would still not make the algorithm fail, though the given path might not be as optimal as original thought it should still give a viable path due to the implementation of the walkAround method.

Some pathfinders have been expanded to foresee other units walk patterns and to take these into their own calculations when finding a route. This possibility do not arise in a world which is not grid-based since the possibilty to 'rent' map space is not available. Unfortunately this option will not be available to a pathfinder not based on the map structuring (ie. grids or another format). In the real world it makes sense not to let all allies know where you are all the time. Implementing a system for units to communicate and plan their movements optimally can be implemented given more time. Currently the walkAround method in the MoveableAI class makes up for collisions. This method should be extended to take unit-to-unit communication into account for smarter move patterns.

## 8.1 Vision

f,g,h? (Be more clear)

[8] fixme: 'The' relax method? Which one?

# Chapter 9

# Unit organization

# Chapter 10

# Unit AI

## 10.1 Hierarchical structure

Most realtime strategy games include two kinds of AI: first there is a simple AI which controls the low-level behaviour of the individual units. This AI is responsible for automatically doing tasks which are trivial, such as firing at enemies within range or, if the unit is a resource gatherer, gather resources from the next adjacent patch if the current patch is depleted such that the player needs not bother keeping track of this. The other kind of AI is the separate AI *player* which controls an entire army, and which is incompatible with the interference of a human player. This AI is responsible for larger tactical operations such as massing an army or responding to an attack.

In JWars, as we shall see, there is no such *clear* distinction between different kinds of AI. Because of the hierarchical organization it is possible to assign an AI to each node in the unit tree, meaning that while every *single* unit does have an AI of limited complexity to control its trivial actions, like in the above case, the platoon leader has *another* AI which is responsible for issuing orders to each of the three or four squads *simultaneously*, and the company leader similarly is responsible for controlling the three or four platoons. It is evident that this model can in principle be extended to arbitrarily high levels of organization, meaning that it will easily be equivalent to the *second* variety of AI mentioned above. The entire army could efficiently be controlled by AI provided that the AI elements in the hierarchy are capable of performing their tasks individually.

There are numerous benefits of such a model, the most important of which we shall list here.

- Tactically, if one unit is attacked the entire platoon or company will be able to respond. In classical realtime strategy games this would result in a few units attacking while the rest were standing behind doing nothing. Thus, this promotes sensible group behaviour which has been lacking in this genre since its birth.
- It is easy for a human player to cooperate with the AI. For example it is

sensible to let the AI manage all activity on platoon and single-unit level while the player takes care of company- and battalion-level operations. This will relieve the player of the heavy burden of micromanagement which frequently decides the game otherwise (as asserted in section **??**). Thus, more focus can be directed on *strategy and tactics* instead of managing the controls.

- The controls may, as we shall see below, be structured in such a way as to abstract the control from the concrete level in the hierarchy. This means the player needs not bother whether controlling an entire company or a single squad: dispatch of orders to an entire company will invoke the company AI to interpret these orders in terms of platoon operations. Each platoon AI will further interpret these orders and have the individual units carry out the instructions.

- A formation-level AI can choose how to interpret an order to improve efficiency. For example the player might order a platoon to attack an enemy tank, but the platoon AI might know that rifles are not efficient against the tank armour. Therefore it might conceivably choose to employ only the platoon anti-tank section against the tank while the remaining platoon members continue suppressing enemy infantry. These considerations are easy for a human player, but cannot be employed on a large scale since the human cannot see the entire battlefield simultaneously. Once again this eases micromanagement.

There are, however, possible drawbacks of the system.

The worst danger of employing such an AI structure is probably that the AI might do things that are unpredictable to or conflicting with the human player. Care must be taken to ensure that human orders are not interfered with, and that the behaviour is predictable to humans[1].

From a game design perspective it might also be boring if the automatization is too efficient, leaving the player with nothing to do. This problem, of course, can be eliminated simply by disabling certain levels of automatization.

## 10.2 Design considerations

It was stated above that the control of single entities versus large formations could be abstracted such that the player did not need to bother about the scale of operations. If this principle is to be honoured, the user interface must allow similar controls at every level of organization. At the software designing level this may be parallelled by providing a common interface to be implemented by different AI classes. It should be possible to give *move* orders, *attack* orders and

[1] Classical examples of this problem are when resource gatherers deplete resources and automatically start harvesting from patches too close to the enemy, or when the player issues a movement order and the unit moves the 'wrong' way into the line of fire because the pathfinder has determined that this longer way is nonetheless faster.

so on, and each of these should have its implementation changed depending on the context, i.e. whether the order is issued to a formation or a single entity.

It is also not entirely clear at this point which operations should be supported. If

interfaces and such

## 10.3 Overview of AI structure

list different AI interfaces

# Chapter 11

# Combat

## 11.1 Analysis of combat dynamics

## 11.2 Weapons, armour and damage

## 11.3 Spotting and targetting

# Chapter 12

# Control

# Chapter 13

# Graphics

# Chapter 14

# Conclusion

# Bibliography

[1] Sean Riley, *Game Programming with Python* (Charles River Media, 2004. ISBN 1-58450-258-4)

[2] T.H. Cormen et al., *Introduction to Algorithms, 2nd Edition* (McGraw-Hill Book Company, 2001. ISBN 0-262-03293-7)