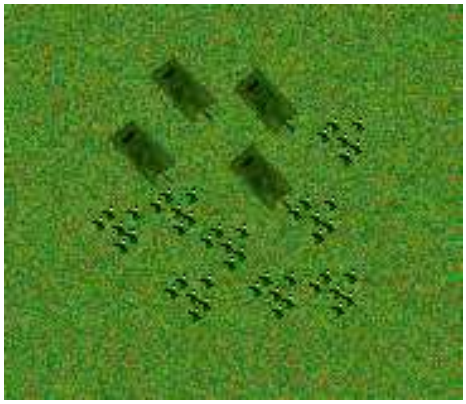
JWars - A Generic Strategy Game in Java

Midterm Project at IMM



Michael Francker Christensen s031756
Ask Hjorth Larsen s021864

Supervisor: Paul Fischer

DTU - Technical University of Denmark

June 8, 2006

This space intentionally left blank

Abstract

Contents

Chapter 1

Introduction

1.1 Acknowledgement

1.2 The realtime strategy genre

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

4.1 Choosing a network model

4.2 Synchronization

4.3 The networking API

Chapter 5

Event handling

Chapter 6

World of JWars

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.

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].

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

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

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 collision detector

blahblah

7.3 Terrain

7.4 Pathfinding

When moving units in the world of JWars a navigational problem arises when finding 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 tilesystem 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 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 $w : E \mapsto [0, infinity]$ 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 convert the dynamic and rather limitless implementation of units and other objects in the world of JWars. We have chosen to a very open approach in the area of unit location and building placement. Any building or unit is placeable anywhere on the map and will not fill out a certain amount of tiles. The option of letting objects take space in the map can not be done in JWars, since the data structures allows object of *any* size in JWars. With the no limitations in size of objects, or the speed the can have, tiles can be partially covered and still count as 'filled' in a pathfinder. This system is however implemented in some games and works well - these games however the above mentioned limitations in object sizes and placements of static objects.

A tile system is incapable of handling the pathfinding in JWars - the aspect of pathfinding on graphs is 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*

[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.

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.

Here is pseudocode for the importent functions in the pathfinder.

```
boolean Shoot-to(Sloc, Gloc)
// When shooting from a vertice to another we will at some point either
// collide with the goal or another object on the map
If(collider = goal)
backtrack
else FindVertices(collider, angle(Sloc, Gloc))

FindVertices(vertice, angle)
// max/min are the points in the collider with the most extreme angle
// values from the original angle
Shoot-to
```

When solving this problem the algorithm only takes into account static objects which are registrered in the as static list in the collision tiles. Unfortunately moving objects has the possibility of making

7.5 Vision

7.6 Data management

Chapter 8

Unit organization

Chapter 9

Unit AI

9.1 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.

9.2 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

9.3 Overview of AI structure

list different AI interfaces

Chapter 10

Combat

10.1 Analysis of combat dynamics

10.2 Weapons, armour and damage

10.3 Spotting and targetting

Chapter 11

Control

Chapter 12

Graphics

Chapter 13

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)