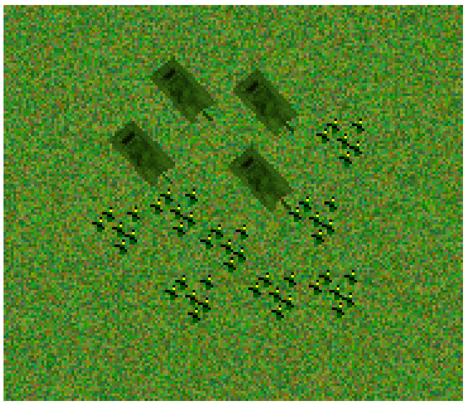
# JWars - A Generic Strategy Game in Java

*Midterm project — Informatics and Mathematical Modelling*

**Authors:**
**Michael Francker Christensen, s031756**
**Ask Hjorth Larsen, s021864**

**Supervisor:**
**Paul Fischer**

August 8, 2006



DTU - Technical University of Denmark
Lyngby

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

## Abstract

This project documents the development of the real-time strategy game JWars which uses a unique hierarchical force structure to improve the player's ability to control large forces, plus provides a basis for advanced AI interactions between units. The game, which is written in Java, includes several API packages providing collision detection, pathfinding, networking and numerous minor components. These are designed specifically for handling large, sparse game worlds and are particularly suited for modelling realistic environments. The report discusses each of the corresponding problems in detail, focusing on analysis and design while demanding little specific knowledge of the Java programming language.

The game aims to incorporate the tactically advanced gameplay of turn-based tactical wargames into a classical real-time setting. The game in its present state demonstrates the applicability of the underlying framework, which provides all the basic functionality required by the genre. Development is expected to continue, adding further game content and complexity.

# Contents

**Abstract** **i**

**Contents** **ii**

List of figures . . . v

List of tables . . . viii

**Preface** **1**

**1 Introduction** **2**

1.1 Introduction to the genre . . . 2

1.1.1 Background . . . 2

1.1.2 RTS combat and control . . . 4

1.2 Why JWars? . . . 7

1.2.1 Flaws in contemporary real-time games . . . 7

1.2.2 Military hierarchy . . . 8

1.3 Overview . . . 8

1.3.1 Features of JWars . . . 9

1.3.2 The JWars API modules . . . 9

1.3.3 Product requirements . . . 10

1.4 Reading this report . . . 11

**2 Architecture** **12**

2.1 Connection and initialization . . . 12

2.1.1 Loading game data . . . 13

2.1.2 Launching the game . . . 13

2.2 Flow of control . . . 14

2.2.1 Synchronization . . . 14

2.2.2 Deterministic behaviour . . . 15

2.2.3 Player input and network instructions . . . 16

2.2.4 AI, orders and their execution . . . 16

2.2.5 Conclusion . . . 17

**3 Networking** **19**
3.1 Choosing a network model . . . . . . . . . . . . . . . . . . . . . . 19
3.2 Synchronization . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
3.2.1 Interactivity: network instructions . . . . . . . . . . . . . . 20
3.2.2 Synchronization instructions . . . . . . . . . . . . . . . . . 21
3.2.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3 The networking API . . . . . . . . . . . . . . . . . . . . . . . . . 22
3.3.1 Implementation notes . . . . . . . . . . . . . . . . . . . . . 23
3.3.2 Random numbers and deterministic code . . . . . . . . . 24

**4 World of JWars** **25**
4.1 Flow of control and timing . . . . . . . . . . . . . . . . . . . . . . 25
4.1.1 Game loop vs. timer . . . . . . . . . . . . . . . . . . . . . . 25
4.1.2 The game timer . . . . . . . . . . . . . . . . . . . . . . . . 26
4.1.3 Game performance discussion . . . . . . . . . . . . . . . . 27
4.1.4 Interpolation during rendering . . . . . . . . . . . . . . . . 27
4.2 Coordinate spaces . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
4.2.1 Coordinate data representation . . . . . . . . . . . . . . . 28
4.2.2 Important coordinate systems . . . . . . . . . . . . . . . . 30
4.3 Terrain . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3.1 Terrain in games . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3.2 Map design . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3.3 Random terrain generator . . . . . . . . . . . . . . . . . . . 32
4.3.4 Terrain objects . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.3.5 Terrain appereance . . . . . . . . . . . . . . . . . . . . . . . 36
4.4 Event handling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.4.1 Types of events . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.4.2 Performance considerations . . . . . . . . . . . . . . . . . . 38
4.4.3 Queueing system . . . . . . . . . . . . . . . . . . . . . . . . 38

**5 Collision detection** **40**
5.1 Basics of collision detection . . . . . . . . . . . . . . . . . . . . . . 40
5.1.1 Divide and conquer approach . . . . . . . . . . . . . . . . . 40
5.1.2 Tile registration strategy . . . . . . . . . . . . . . . . . . . 41
5.1.3 Shapes and sizes of colliding entities . . . . . . . . . . . . 42
5.2 Design of the collision detector . . . . . . . . . . . . . . . . . . . . 43
5.2.1 The checking routine . . . . . . . . . . . . . . . . . . . . . . 43
5.2.2 The collision grid . . . . . . . . . . . . . . . . . . . . . . . . 44
5.2.3 Further features . . . . . . . . . . . . . . . . . . . . . . . . 44
5.2.4 Efficiency and optimization . . . . . . . . . . . . . . . . . . 45
5.2.5 Using the collision detector . . . . . . . . . . . . . . . . . . 45
5.3 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**6 Pathfinding** **47**
6.1 Pathfinding in general and in JWars . . . 47
6.1.1 The algorithm . . . 48
6.1.2 Data structure . . . 50
6.2 Implementation . . . 51
6.2.1 Expanding and searching . . . 51
6.3 Final design . . . 55

**7 Dynamical game objects** **58**
7.1 Unit organization . . . 58
7.1.1 Real-world military organization . . . 58
7.1.2 Military command in computer games . . . 59
7.1.3 Tree-based unit representation . . . 60
7.1.4 Network distinguishability of units . . . 61
7.2 Game data management . . . 61
7.2.1 Inheritance versus data-based game object classification . 62
7.2.2 Category model . . . 62
7.2.3 Content loading by categories . . . 64
7.2.4 Conclusion . . . 65
7.3 Unit AI . . . 65
7.3.1 Hierarchical structure . . . 65
7.3.2 Design considerations . . . 68
7.3.3 AI layering structure . . . 69
7.3.4 Future AI work . . . 69

**8 Combat dynamics** **71**
8.1 Firing and damage . . . 71
8.1.1 Combat rule set . . . 72
8.1.2 “Weapon vs. armour”, or “armour vs. weapon”? . . . 73
8.1.3 Structure of the weapons API . . . 74
8.1.4 Firing routine . . . 75
8.1.5 Impact handling by armour . . . 75
8.1.6 Conclusion . . . 76
8.2 Vision . . . 76
8.2.1 Vision in games . . . 77
8.2.2 Approach 1: direct observer-observer checking . . . 78
8.2.3 Approach 2: observer-terrain checking . . . 79
8.2.4 The spotting routine . . . 79
8.2.5 Final design . . . 81
8.2.6 Evaluation and discussion . . . 82

**9 Graphics** **83**
9.1 Active versus passive rendering . . . 83
9.2 Double buffering . . . 84
9.3 Battlefield rendering and layers . . . 85
9.4 Optimization of the rendering routine . . . 86

9.5 Conclusion . . . . . . . . . . 87

**10 Game improvements** **88**
10.1 Future work . . . . . . . . . . 88
10.2 Known issues . . . . . . . . . . 89

**11 Conclusion** **90**

**References** **92**

**Appendix** **92**

**A Game manual** **93**
A.1 Running the program . . . . . . . . . . 93
A.2 In-game control . . . . . . . . . . 95
A.2.1 Using the panels . . . . . . . . . . 95
A.2.2 Marking and moving units . . . . . . . . . . 97

**B Development plan** **100**

# List of Figures

1.1 Screenshot from Starcraft. The voracious Zerg swarm is overrunning a Terran settlement. . . . . . . . . . . . . . . . . . . . . . . . 6

2.1 Flowchart illustrating the process of executing an instruction issued by a player. The player performs an action which results in the networking code writing the appropriate instruction identifier and data across the network (dashed line). The server adds a time stamp to the instruction and echoes it (second dashed line) to all clients, at which point the instruction is scheduled for execution at the specified time. . . . . . . . . . . . . . . . . . . . . 17
2.2 Schematic overview of the flow of control. Black arrowheads denote that one module affects the state of another actively, whereas white arrowheads denote a flow of information from one component to another (requested by the component towards which the arrow points). New information enters the system only via networking. The flow of information from the world to the rendering code encompasses all unit positions, in reality this is available through the collision detector. Not included on the graph: firing can lead to elimination of units which in turn removes them from the collision detector and results in the invocation of relevant AI methods. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18

4.1 Two coordinate systems. Axes are similar to those normally used with screen coordinates. The black × in the left system is transformed to the × in the right system, but the inverse transformation yields the grey × in the left system because of integer division. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
4.2 The mid-line displacement used on a one-dimensional map . . . . 33
4.3 The diamond square algorithm running on a 5x5 grid until termination. If the grid were larger, the first steps would be identical, but the algorithm would continue by halving the *step* size and performing the same operation over and over. . . . . . . . . . . . 34
4.4 This is a 3D model of a diamond-square algorithm running on a NxN map. It clearly illustrates how jagged maps come to look more natural. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.5 These are two maps randomly generated by the terrain generator. It is not immediately noticable but both maps are periodic, i.e. their edges wrap. . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.6 Terrain graphics. Though the terrain consists of tiles, 32 pixels on each side, this is not clearly visible due to the amount of varying tile images. Some trees are visible in the right side of the picture. 37

5.1 The collision grid visualized. The number of units registered in each tile is listed inside the tile. This is an in-game screenshot; the debug grid can be enabled by passing -d as a runtime parameter. 42

6.1 A simple pathfinding problem. . . . . . . . . . . . . . . . . . . . 49
6.2 A single iteration in the loop of the pathfinder. The expand function calls it self repeatedly so all needed nodes are found. . . 54
6.3 The illustration shows the pathfinder tracking around the large object on its way to the target zone. The fastest route however is to ignore the large objects and go straight for the smaller building, around, and then for the goal. . . . . . . . . . . . . . . . . . . . . 56

7.1 Example of a unit tree. Only the nodes with downward pointing arrowheads are expanded. This is part of a screenshot from JWARS. 60
7.2 Different unit classes by inheritance hierarchy. . . . . . . . . . . . 63
7.3 Parallel inheritance hierarchy of unit classes in JWARS and category classes. The fully inked arrows denote inheritance relationship, while the dashed lines denote correspondence between a class of unit and a class of category. . . . . . . . . . . . . . . . 64
7.4 Categories. The continuous boxes indicate category classes whereas the dotted boxes list examples of actual category objects of the corresponding class. Arrows indicate inheritance. . . . . . . . . . 65

8.1 Soviet T-34 tank with wire mesh for protection against the Panzerfaust anti-tank weapon.[9] . . . . . . . . . . . . . . . . . . . . . . 74
8.2 Armour statistics for a T-34 tank. This is part of a screenshot from the game. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
8.3 Side hit or front hit? The projection **S** of the tank side on the normal of the impact direction is about equal in length to the projection **F** of the front, so the probability of hitting the side is about 50 %. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
8.4 The spotting routine. The current observer is surrounded by a dashed circle, the radius of which is equal to the maximum vision range. The algorithm will traverse the four tiles that overlap the circle and expend no time checking the rest of the map. These four tiles contain eight observers aside from the current one, four of which are outside the vision range. The remaining four observers (excluding the current one) that are inside the vision range are subjected to an actual observer-observer check. . 80

9.1 The rendering routine. Different steps are indicated by numbers. Steps 4 and 7 are very fast on modern computers and are not likely to have significant impact on performance. . . . . . . . . . . 87

A.1 The JWars launcher. . . . . . . . . . . . . . . . . . . . . . . . . . . 94
A.2 The game lobby. Manstein and Rokossovsky are chatting before a friendly game of JWars. . . . . . . . . . . . . . . . . . . . . . . . 95
A.3 The battle is raging between the Russian and German forces. . . 96
A.4 A green circle around units is a notification about the players current selection of troops. When troops are selected their destination coordinates become green lines on the ground to illustrate their current heading. . . . . . . . . . . . . . . . . . . . . . . . . 98

# List of Tables

7.1 The datafile entry defining the weapon category corresponding to a German 75mm Kampfwagenkanone (tank gun). The right column contains the actual lines in the datafile, while the left column is only for description. The firepower data comprises ammo type (armour piercing), armour penetration (in millimetres) and “kill index” (effectiveness against infantry). . . . . . . . . . . . . 66
7.2 Datafile entry defining the German Panzer IV tank. The entries in the weapon list are identifiers of weapons. Notice the identifier of the tank gun from Table 7.1. The other guns and the hull and turret types are also identifiers of categories. These include filenames of images which are used to display the components. . 66

8.1 Equivalent terms of collision detection and observation handling. 81

# Preface

During the development of JWars many friends have taken the time and trouble to test the code on many different platforms and hardware. This help has been of immense value to us, particularly for testing the graphical performance using different drivers and graphics adaptors, not to mention the performance of the networking code under less-than-optimal (non-LAN) conditions. In particular we would like to thank Dennis Dupont Hansen, Kasper Reck, Peder Skafte-Pedersen and Kenneth Nielsen.

Finally we are very grateful for the help and patience of our supervisor Paul Fischer with whom we have had numerous technical discussions about the various software components.

# Chapter 1

# Introduction

## 1.1 Introduction to the genre

This section is meant as an introduction to the *real-time strategy* (from now on also known as RTS) genre. This section should be seen as history of the genre as well as a opportunity to understand the general game structure and the more advanced concepts in the genre. First we will define the RTS genre and then a quick walkthrough around it's history. In the end we will point out the importent features implemented in RTS games over the years. These features will be importent for our project since our goal is to develop a game which engine live up to the time's standard.

### 1.1.1 Background

JWARS in its present form is in the most technical sense of the words a *real-time tactical* game. The term strategy applies to large scales of operations where logistics and supplies become headaches. JWARS models combat at the battalion level while excluding base building and resource gathering which are otherwise normal features in RTS games, and is by some definitions therefore merely a tactical game. Even so, the scale of operations is ironically *larger* than in most real-time strategy games, and we shall therefore amidst the controversy of genre definitions take the liberty of categorizing JWARS as an RTS.

The RTS genre came about in the 80's, but was only fully developed and formally seen as a single unique genre 10 years later with titles as Westwood's Dune II and Blizzard's Warcraft and Warcraft II. For the casual gamer an RTS game can be recognized by some simple properties which have grown to distinct the genre:

1. War planning is essential – strategy

2. The player has no 'Next turn' button, but instead time progresses continuously – real-time

Other typical properties:

1. Resource gathering and management
2. Base building and army production. The army consists of *units*, controllable entities which can fight.
3. The player has direct control of his units/buildings
4. The player must defeat the opponent(s) in battle

The RTS genre was developed from the turn-based strategy games genre. One of the first RTS games, perhaps the most defining game for the genre, is Dune II whose developers were inspired by Sid Meier's 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. As such, many diversities have risen in the RTS genre as game developers become more inventive. Today RTS games are in general built on a player vs player environment yet providing single player campaigns consisting of pre-defined scenarios where the player fights the computer.

Most strategy games require the player to understand basic military concepts and most often a paper-rock-scissor approach on unit combat. A unit can defeat some opposing units, while it in turn will be defeated by a suitable opponent unit. Often this is combined with gradual unit improvements by development in the player's armoury for the cost of resources and time. Resources are mentioned as a basic concept in RTS games since economy leads to more higher military power which in turn leads to higher resource income either by conquering land or holding strategic resource areas. This has been the basic approach to strategy games, gather resources, build up military forces, gather more resources or focusing on cutting off the opponent's supplies and destroying enemy resource areas. In this cocktail of choices for the player comes the tactical manoeuvres and structural placements if possible. Most games today try to incorporate terrain as a factor in the games and many aspects of real warfare has come in to play like high ground, bottleneck manoeuvres, entrenchment and so on. As the computer game industry grows, so does the amount of time and money spent on developing new features in strategy games. Many of the more succesful games found a firm middleground in supporting a lot of features but not making the game dependent on these. This will allow more simple users capable of enjoying the game in a relaxed playstyle while the hardcore gamers can dive into micromanagement[1] of troops, exploitation of game engines etc.

The average RTS game normally uses the single player campaigns as a linear story introducing sequentially more advanced units/concepts along the story. Often a campaign starts with the player only controlling few simple units with few degrees of freedom for the player as the mission is laid out. As the player

[1]Micromanagement refers to the player ordering each specific unit around to optimize their performance - as opposed to macromanagement which involves larger troop movement, maintaining production and similar issues.

completes missions more units and buildings or concepts will become available - in this way a new bought product will introduce units slowly and let the player familiarize himself with the game features in turn, thus not making the game seem too complicated. In the JWARS project however we will not be including single player missions as we would rather spend time developing the engine than setting up specific scenarios.

In the last couple of years RTS games have been improving greatly in one specific area - graphics. Most of the popular older games relied on 2D graphics while the 3D environments in first-person-shooters blossomed. Not until Blizzard's release of Warcraft III: The Frozen Throne did it become a standard to use 3D engines in RTS games, though earlier games using 3D graphics had been around for a while without conquering large market shares. Graphics influenced some games' popularity, though most is based on gameplay and the universe in which the game takes place. Almost all newer titles use a 3D engine with changeable view angles and zoom function, in this project however we rely on 2D graphics and focus on gameplay and the gameengine itself. This is not *purely* something we do to save time: 3D environments can become confusing, meaning the player's ability to control his forces suffers.

### 1.1.2 RTS combat and control

RTS games focus on large scale combat. All actions made by a player are primarily made with the thought of increasing his force strength. With this in mind an example of *unit balancing* and a brief explanation of a GUI will open some doors for the inexperienced players.

In RTS games the player should be able to choose between a wide selection of possibilities for combining his forces. This is where unit balance and the strategy idealism creates synergy and creates the dynamic atmosphere in which the genre unfolds its true gameplay. The term *unit balance* is used to determine an ordering of how units compare against each other in combat.

In this instance we generalize the concept for better understanding. If we create an example with 3 different units being measured against each, other for instance an aeroplane, a tank and an anti-aircraft gun (AA gun), logic would create simple rules for this setup:

- Plane beats tank
- Tank beats AA gun
- AA gun beats plane

We could attach a force ratio on each instance if we wanted to use a measurement of how many of one type it would take to defeat the other type. This looks like a standard rock-paper-scissors setup and a player would never be able to select a single strategy and be sure to win. By expanding this theory into containing more different units with strengths and weaknesses the tactical gameplay is ensured in the game as the players will need to take steps countering each other throughout the game. Normally a player can choose between wide varieties of units to counter out the opponents units. This would normally create a stalemate for two armies fighting. If however the one side has access to a unit which counters most or all of the opponents units this would destroy the balance of power, thus making the unit too effective. By creating a unit which is overpowered in this way players could ensure a higher chance of victory than normal by using this unit extensively. When games contains such units it becomes unbalanced and require unit balancing. The unit balance can be compromised by several factors as each attribute needs to be balanced out against other units i.e. the more complex the game the harder to balance.

The real idea behind unit balancing is not to have a units strength on a linear model, but let attributes like speed, length and accuracy create units suited for special situations.

Unit balancing is one of the greatest challenges for developers and is often an ongoing process even after release. Games today which base their playerbase on an online environment have the ability to release updates when needed. Mostly the developers will release a game which is unbalanced unintendedly, and only the testing done by players when playing the game will find the issues which need attention. Some developers has adopted the theory that there is no testing like releasing the game to a massive audience.

Next we will introduce one of the most classic games in the genre as an example of how a game GUI could be created. The example we have chosen is Blizzard Entertainment's *Starcraft* including the expansion pack - *Starcraft: Brood War*. This game has been chosen because it is a well-known[2] and typical example within the genre, and because both authors of this paper are proficient in this game.

Most RTS games have extremely similar user interfaces. Several designs with unique abilities and setups have come up but invariably contain the two most common components – a *main display* or *focus panel* which focuses on a limited part of the battlefield and displays nice and detailed graphics of the action, and a smaller *minimap* which shows the entire map but only conveys little information. Add to this any number of status- or control panels. These are all tools for the player to enhance his control and provide important information.

Figure 1.1 shows a screenshot from Starcraft. As usual, the GUI is split into different components, each providing the player with information and options. Covering most of the screen is the focus panel.

The player can *select* units displayed here by dragging a box around them using the mouse. The player can now give orders to this selection. The exact way to give orders differs between games. In starcraft, right-clicking on the ground, for example, will cause the currently selected units to walk to the specified location.

The minimap is located on the bottom left corner[3]. The minimap serves as

[2]*Starcraft* is for example extremely popular in South Korea, where public competitions are regularly televised and famous players are sponsored.[5]

[3]The bright dots on in the minimap is the players base while most of the map is unexplored – black. Notice the rectangle on the minimap showing where the focus panel is centered. Another detail is the large rectangle in the minimap which indicates that the minimap doesn't

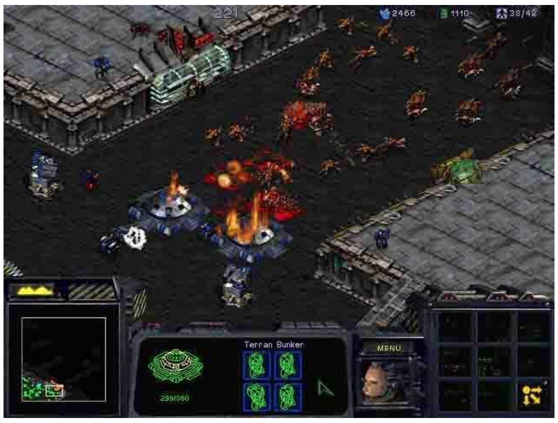


**Figure 1.1:** *Screenshot from Starcraft. The voracious Zerg swarm is overrunning a Terran settlement.*

a primary overview for the player to switch his focus on the battlefield. The minimap usually shows the player's own forces in green and opponents forces in red. In this way an enemy massing forces or approaching your territory will result in red markers on the minimap. The minimap will never be the player's main source for information as the information it provides is always sparse and can even be misleading.

The lower middle panel provides information about the currently selected objects on the battlefield. When a player has his focus on a specific unit or object all relevant information concerning the object will be displayed here. This is the most direct information the player can get from the game as it will often display a single unit's statistics like firepower, range, speed and health status. In the referred screenshot a bunker is selected showing its current health status and an amount of marines occupying it.

Finally, the lower right panel contains controls that are available for the current unit selection. Since the current selection is a bunker, which is immobile, it only has one control button (though it normally has more): pressing it will cause the marines inside to leave.

The user interface in Starcraft is a standard example for the genre. The simple GUI handles most situations very well and this setup is used by most RTS games today. Most new players to an RTS game have a tendency to use the focus panel as the only source for information while watching the minimap is in reality extremely important, for example allowing the player to spot enemy attacks earlier.

stretch to fill the entire panel as it does in newer releases.

## 1.2 Why JWars?

This section introduces JWars and why the authors believe this is worthy of a project. First we shall consider some flaws or features absent from contemporary games, then we shall see how these might be remedied.

We have chosen this project with a particular purpose in mind – to create a game which combines the realistic tactical elements of *turn-based tactical games* and the fast pace of *real-time strategy games*. Where the turn-based games never stress the opponent and give him arbitrarily long time to make a decision, they rely on realistic features and complicated combat systems. These elements have not been seen in any mainstream real-time games yet, as RTS game engines rely on faster paced gameplay on smaller maps thus excluding realistic distances and other game content. This concept in itself is not exactly new, but shortly in Section 1.2.2 it will become clear what separates JWars from the hitherto existing games.

In developing JWars we want first prove that the combination of the game types is not unrealistic. This is done by developing the necessary API packages that are supposed to allow other developers to continue our work, since we will surely not be able to finish an entire game of commercial quality ourselves within the time limit of this project.

### 1.2.1 Flaws in contemporary real-time games

There are some areas in which the real-time strategy genre has not evolved much over the years. Some of these are:

- Individual units typically behave unintelligently unless the player takes care to control each (or very small groups) of them personally. For example, if an enemy approaches a group of friendly units then half the group might attack and be lured into an ambush whereas the other half stays idle. Also it is frequently observed that anti-tank weaponry will be automatically directed at infantry even though enemy armour is nearby as well.

- As the game progresses, complexity grows greatly as units are produced, and the player cannot hope to control forces with such attention to detail. This directly benefits the player who is quickest with a mouse or keyboard, and not the player with superior strategic ability. Control, rather than strategy, thus becomes the primary point of concern during gameplay.

- While not necessarily a drawback, most games use *hit points* (this is discussed in Section 8.1) to represent a unit's health. When damaged, some hit points are deducted until the hit point count reaches 0 at which point the unit in question dies. Thus most games are deterministic in nature, or contain only negligible random factors in combat.

### 1.2.2 Military hierarchy

Many of the drawbacks pointed out above can be eliminated by introducing a *tree-based* means of controlling units. Such a system is in reality a requirement of *any* working military as we can clearly see in the world today, and it is therfore curious that no attempt has yet been done to incorporate such a system in real-time strategy games.

Aside from easing the control of large forces for the *player*, it is possible to provide better AI support using this system. By using a tree hierarchy in the game, a simple AI can be assigned to every military formation "leader", such that this AI is responsible for controlling the immediate subordinate formations. The flat unit structure in most real-time strategy games allows for little organized interaction through unit AI, but by explicitly embracing a military structure, multiple platoons and companies can work together, controlled by automated commanders.

The AI-specific possibilities implied by this system are almost endless, yet bearing in mind the time necessary to develop such a system we can hardly hope to achieve any impressive results in this field since the entire game has to be built from scratch. What we can do, however, is to provide API components that demonstrate the applicability of this model, and therefore opens the way for future development of the AI.

The increased controllability obtained by using a tree-based hierarchy allows players to control nearly arbitrarily large forces. Consequently it can be expected that focus on tactics will become relatively more important.

## 1.3 Overview

The software presented in this report can roughly be diveded in two sections: the JWars *game* (or just JWars) and the JWars *API*, or application programming interface, which are both written in the Java programming language. The game is in reality a thin shell of specialized code – comprising user interface and control – plus the game content, which works on top of the API packages that are responsible for handling more complex problems.

The JWars API consists of several modules which can be used separately or with a minimum of cross-package dependencies. The following chapters will describe each of these modules in turn, but in order to achieve an overview, we shall list the main modules briefly below. Chapter 2 is devoted to describing their high-level interaction in the game. The important API modules are largely feature complete.

The game itself represents a genuine effort of creating a quality piece of software and does not only serve as a means of demonstrating and testing the packages. However game development is time consuming and normally involves much larger teams of programmers and designers. Therefore in its present state the game, while fully playable, includes only the most important features, and has not yet been balanced for "serious" play. Most of the required work on the game is of relatively trivial nature and does not hold any technical problems worth mentioning.

### 1.3.1 Features of JWars

The game is a fully playable two-dimensional top-down view real-time strategy game. The game takes place on the eastern front in World War II, and the available weapons correspond roughly to the situation in the fall of 1943, at the Battle of Kursk. Many ideas are borrowed from advanced turn-based tactical games which are not normally seen in real-time games, making it unique and different from contemporary games. Features include:

- Two teams: the German Wehrmacht and the Soviet Red Army. Each side possesses roughly two battalions worth of tanks and infantry.
- Supports multiplayer over LAN or the internet. Opposing players are expected to control the two forces. Additionally, several players can share control of each force simultaneously for cooperative play.
- Opposing forces automatically fire at each other. Combat dynamics are highly realistic, using e.g. real-world tank armour tables.
- Explicit military hierarchy allows efficient control of large military formations.
- A large game world allows players time to focus on tactics.
- Beautiful (but simple) randomly generated graphics.

### 1.3.2 The JWars API modules

This is an overview of the generic software components that can be reused in other games.

- 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. The scaleable 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 and updates the search area according to requirements. This approach accomodates obstacles of arbitrary size and placement, and accomodates large maps without excessive memory footprint.
- 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. Every unit and every formation has an AI which is responsible for interpreting and carrying out orders. The present AI implementations are still very simple, but the framework is designed with extensibility in mind.

- 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 a data repository and organized in *categories* which serve as *factories* for different unit types.

- 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 strict logical synchronization between clients and server.

- Multiplayer synchronization utilities. Synchronization on multiple clients is done by means of a timer which assures that clients follow the server temporally closely.

- Rendering routine. The display is actively rendered using *double buffering*, supplied by an extra backbuffer which used to reduce the repaint count of static objects such as terrain.

- Terrain generator. The terrain generator creates random continuous maps which can be used as e.g. altitude maps.

### 1.3.3 Product requirements

Before starting the project we had some minimal requirements which would have to be done within the project's time limit. At the beginning of the project we wrote down the minimal requirements. The minimal requirements were listed but has been rewritten to this:

The game must be playable over the internet by at least 2 players. For game content we must have a working GUI making the player capable of giving orders and gathering information. Units must be able to move around in the world and shall automatically begin firing at opponent units within the field of sight. Units must be able to sustain damage as well as being destroyed and be removed from the game. The world must provide different terrain types which should be able to have an effect on the units occupying space within the the given terrain type – like movement speed or visiblity alterations.

## 1.4 Reading this report

The current chapter – Overview – should hopefully have provided a clear idea of which modules we have worked with and what the game is like.

The following chapter – Architechture – is meant to provide actual insight in the workings of JWars, the way different modules interact with each other and are glued together by the game code.

The main part of the report follows, and here the different modules will one after another be discussed and evaluated in great detail. Note that in most cases we have avoided explicit code and language-specific information, preferring a higher level of discussion focusing on analysis, design and algorithms. The report should thus be of interest to game programmers and not only Java programmers.

The structuring of the report is loosely chosen such that modules with few dependencies (for example networking and world representation) are treated first, and the subjects progress to successively higher levels of abstraction and interdependence in the later chapters. Still, the chapters should be independently readable.

In the end of the report is the Appendix containing the *Game Manual* which is useful for running and using the application.

# Chapter 2

# Architecture

In this chapter the architecture of JWars will be described, i.e. the way in which the different components are made to interact. It should be outlined that the descriptions in this chapter are kept *brief.* There are far more operations under the hood that noted here, but it would be too cumbersome to describe the less important routines. This chapter will only mention the *most important* steps. The subsequent chapters will then go into greater depth describing how the individual components are designed.

## 2.1 Connection and initialization

As the program is started, a small GUI is presented which allows the user to create a server or join an existing one. If the user wants to join a game, this will spawn a JWars *session* which attempts to connect to the specified server.

Creation of a server will always result in a client being spawned locally which connects to that server so as to allow the server's user to participate in the game. This client is no different from any other client (connecting from remote), even though it is physically running in the same virtual machine as the server. The client thus runs independently of the server, but the server uses some common functionality of the client, such as the timer and network instruction set. The practice of giving the server access to the logic of the local client also allows the server to check the validity of orders issued (this has not been implemented, but this is one reason for choosing the design) by the players before relaying that information to the clients. This reduces the possibility of cheating.

When a client session is spawned, the first thing done is to connect to the specified server whether it is local or remote. This allows the client to receive initialization data from the server, such as a random seed and the size of the map to be played[1].

[1]For reasons of debugging, the random seed is always 0 in the current implementation, and only one map will presently be generated, but the order of initialization allows for dynamical specification of game data.

### 2.1.1 Loading game data

After connecting, the game world is generated. This involves a number of steps, namely creating coordinate systems and tile representations of terrain, along with the creation of a collision detector and an *observation environment* (which is responsible for checking whether enemy units can see each other on the map). Notably this step also involves registering the *root unit*, which is the ancestor in the tree hierarchy of all units (see Section 7.1.3) which will later be added to the world.

The following step reads all unit, weapon and formation data from external files (though this could easily be done through the network as well). This kind of data storage is obviously preferable to hardcoding; in fact it allows people to change the game content completely without looking at the source code, by entering data in a simple script-like fashion. This information is represented in *category* objects, which hold data pertaining to specific types of units. For example, the information of a Panzerkampfwagen IV is read once, and then scores of panzers can be spawned using the category as a factory and data storage.

The game presently adds two German and two Soviet battalions to the game, and places them in pre-determined wedge-like formations (this is hardcoded since implementing an entire editor, which is the "normal" way to do this, would take too long) in opposite corners of the map. The battalions are organized in companies and platoons, containing both tanks, assault guns[2] and infantry.

The final step is to build the main Swing GUI which will be displayed during the game. Even though the game is not yet about to start (clients are still joining the server) it is preferable to generate the GUI now, such that the GUI is ready when the game is started.

### 2.1.2 Launching the game

At this point the entire game setup has been loaded, but the game has not yet started. Rather the person hosting the server will want to wait until a enough clients have joined (even though this game only has two armies, several players can control the same army to increase efficiency), and meanwhile a list of the currently connected players is shown, displaying the player names and which army they control. This *lobby* frame is also equipped with a chat for convenience.

The game starts when the server presses the *launch* button. This will result in a *launch* instruction being sent to all clients. When received, it will dispose of the lobby frame and start the timer which controls the flow of time (in the game). It will also make the main GUI visible. At this point the game is fully running, and will remain in this state forever or until the players quit.

[2]An assault gun is a gun mounted on a tank chassis but without a traversing turret

## 2.2 Flow of control

Most real-time computer games run by means of a *game loop*, i.e. a loop in which each iteration constitutes an update of the game state and display as quickly as possible. JWars, too, runs by continuously applying updates. However, in order to ensure that the clients run equally fast, the update rate is instead fixed by the previously mentioned timer. The timer executes those updates from the AWT/Swing event dispatch thread, which means no synchronization with the Swing-managed display is necessary. However the timer also provides the possibility of using its own thread, which might be desirable in non-AWT/Swing games.

The timer attempts to adjust the game flow to that of the server. If an update is completed before it is time to perform the next one, the timer will sleep for the appropriate amount of time before invoking the next update. But if the game flow lags behind that of the server, for example because the computer is too slow to perform updates at the required rate, the timer will report its concerns by passing parameters to the update routine, which will take note of this and attempt to regain lost time by skipping non-vital parts of an update. This brings us to the next point, namely the basic components of such an update.

One update consists two steps.

1. The game logic is updated. This means that all units are updated, allowing them to move (using the collision detector), turn around, take aim, fire and so on, depending on their destination or target. Actually this is the result of the *update* method of each unit being invoked recursively down the unit tree. The logical update will also include various other tasks, such as polling for network input and input from the keyboard. Importantly, this will also poll the *task scheduling system* which stores and manages tasks that should be performed after a delay.

2. The primary graphical display is updated[3]. This involves redrawing any parts of the terrain on which there are moving entities (if no moving entities are nearby the terrain is not redrawn since no changes have happened), then drawing all the visible entities.

### 2.2.1 Synchronization

In case the timer is lagging behind schedule, for example due to the local computer not being able to run the game at the required speed, the graphical update will automatically be performed only a few times per second (such that the display still appears responsive to the user) while logical updates will be performed at the maximum rate possible for the CPU. This means a computer will have to be very slow in order not to be able to play the game. It also means that if one computer is slow, it will not delay the server and the other clients (a

[3]There is a number of other graphical side displays which are not updated continuously here, but instead by regular AWT/Swing repaints.

problem which is noticed immediately in certain games such as *Command & Conquer: Generals*), but it will be responsible for regaining the lost time itself by sacrificing graphical smoothness in the meantime.

In order to ensure that clients do not execute updates too quickly such that instructions from the server arrive too late (and thus bring the game out of synch), the client continually receives *synchronization instructions* from the server which specify the amount of updates the client is allowed to perform. In the event that the client cannot proceed executing updates because it receives no synchronization instructions from the server, it pauses the timer and waits for new instructions. As soon as the new instruction is received, game updates will be executed at the maximum possible rate until the game time is consistent with the real time elapsed. This means the game will stay in synch during *lag spikes* (small periodes of exceptionally high response times) or even if the player accidentally rips out the cable for a moment.

### 2.2.2 Deterministic behaviour

For the moment we shall ignore the activity of players and concentrate on the tasks performed deterministically as time progresses. There are some operations which are not desirable to do from the main update routine, i.e. those things that do not happen *all* the time. For this reason there exists a framework for scheduling tasks to be performed after a certain delay (such a framework is not strictly necessary since anyone could use if-sentences and countdowns from the main update method, but such approaches would be cumbersome and inefficient). Reloading of weapons is managed in this way: when a weapon fires, it schedules a reload event which will in turn be executed at the proper time, allowing the weapon to fire again.

Another problem is determining which units can see enemy units. This is relatively demanding, because large amounts of terrain may have to be traversed to perform such checks. An *observation environment* takes care of traversing the relevant terrain efficiently. For each *observer* registered in the observation environment, such a check is performed regularly, and the frequency of these checks is controlled – once again – by using the event scheduling framework. The unit AI uses the spotting checks to update targets: whenever a new target is found which is closer than the current one, the unit will automatically focus on destroying the closer target.

When a unit has a target, it will aim its guns towards that target and fire the guns whenever they are ready. When the target is destroyed, it will acquire a new target and continue. When a weapon is fired, the game will randomly calculate a hit location, find units near that hit location and finally calculate the damage done to those units. Infantry and tanks act differently to incoming fire; infantry units can take a random amount of casualties based on the volume of fire, whereas tanks use a more advanced (and realistic) model, taking into account armour plate thickness and slope (using historically correct values), the ability of the weapon to penetrate armour, and several other things. When a target is destroyed, it will be removed from the vision model and collision detector, but technically it is not entirely removed from the game. It still sits in the unit tree, though it is now counted as a *casualty*.

Finally there are some updates to the GUI which are performed at regular intervals (using the event scheduling framework). For example the score board updates casualty and force strength tallies as the game progresses, and the minimap is updated regularly.

### 2.2.3 Player input and network instructions

The location of the main display on the battlefield is managed through the *viewport*. When the viewport is moved (there are multiple ways to do this), it will alert any registered *viewport-event listeners*, which ensures that the view is updated correctly. The player can *select* units using the mouse, and this is managed similarly by alerting a number of registered *unit selection listeners* which can react by updating displays to convey information about the newly selected unit. Unit selection and viewport scrolling are the only non-trivial client-side controls.

Suppose the player presses a key or uses the mouse. Either this action affects the local client only – for example, if the action is just scrolling the viewport across the battlefield, it can be resolved locally.

If, however, the action issues an *order* to one of the player's units, it is necessary to send that instruction across the network. The appropriate instruction will therefore immediately be sent to the server, which will relay that information along with a *time* stamp – information about when exactly that order should be executed – back to all the clients. When the clients receive this instruction it will be enqueued, using the event scheduling framework, until its execution time which the server specified. Finally, when the time is up, the instruction is interpreted and carried out (technically by invoking one of its methods: the instruction is responsible for executing itself). This process is shown schematically on Figure 2.2.3.

### 2.2.4 AI, orders and their execution

Each unit, being either a formation consisting of several sub-units (such as a company containing several platoons) or a single physical entity such as a vehicle, is equipped with a simple AI. Whenever a player issues an order to a unit (such as a move order), and the networking framework has distributed it correctly on all clients, the unit's AI will interpret the order and execute it accordingly. For example, if the unit is a formation it will pass on a move order to its sub-units, making sure that the sub-units receive different destinations such that they line up in an orderly wedge-like fashion (similar to the initial setup mentioned in Section 2.1) instead of having all of them try to reach the same point. If the unit is not a formation but instead e.g. a vehicle it will invoke the pathfinder to calculate a feasible path towards the destination (consisting of a number of waypoints), then simply register its new destination and begin moving towards it at every update (as described in Section 2.2).

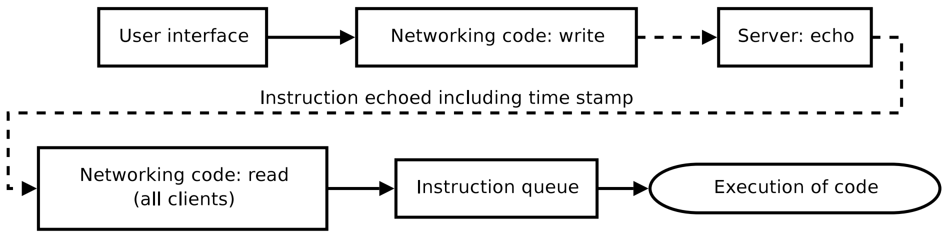


**Figure 2.1:** *Flowchart illustrating the process of executing an instruction issued by a player. The player performs an action which results in the networking code writing the appropriate instruction identifier and data across the network (dashed line). The server adds a time stamp to the instruction and echoes it (second dashed line) to all clients, at which point the instruction is scheduled for execution at the specified time.*

While orders can be given by the player, it is possible for different AIs to give orders to each other. This happens when a formation AI is passing on a movement order to its sub-units, but in a broader perspective (not yet implemented functionality) this can be used to achieve sensible interaction between elements of the same formation, ensuring e.g. that all platoons of a company attack together properly, or that they wait together in an ambush without firing before the time is right.

### 2.2.5 Conclusion

Having read this, you should understand how the different components in the game interact at a high level. The entire situation is illustrated on Figure 2.2. The rest of the report is devoted to explaining the individual components, analysing their requirements and deriving proper designs.

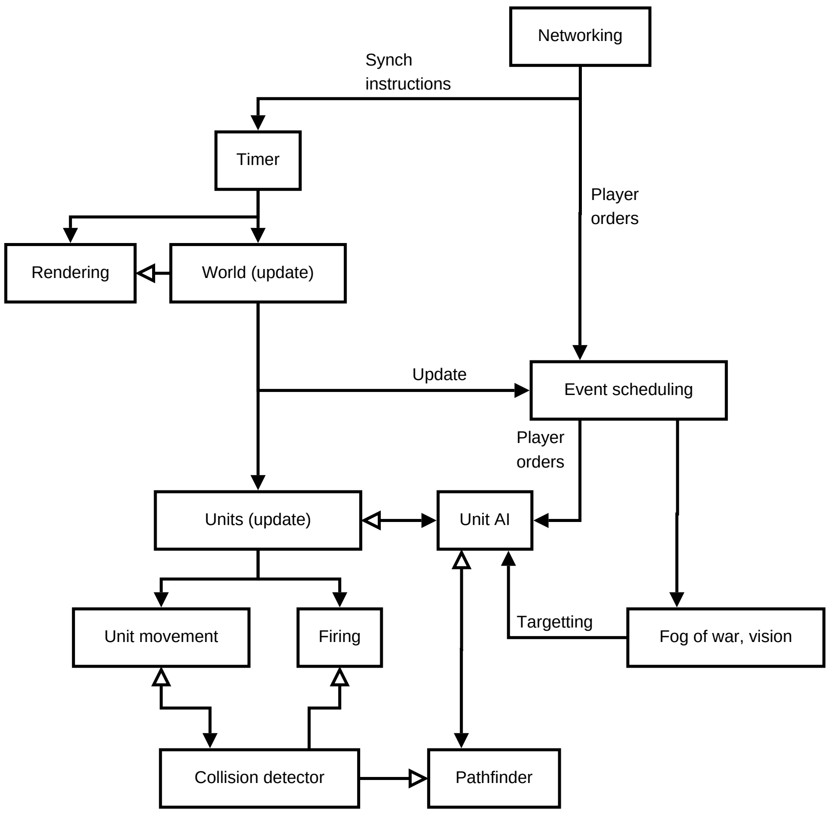


**Figure 2.2:** *Schematic overview of the flow of control. Black arrowheads denote that one module affects the state of another actively, whereas white arrowheads denote a flow of information from one component to another (requested by the component towards which the arrow points). New information enters the system only via networking. The flow of information from the world to the rendering code encompasses all unit positions, in reality this is available through the collision detector. Not included on the graph: firing can lead to elimination of units which in turn removes them from the collision detector and results in the invocation of relevant AI methods.*

# Chapter 3

# 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 solution can have profound impact on this[1]. This chapter will explore the options available and in turn decide on a feasible design.

## 3.1 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: first there is the *amount* of game data which has to be synchronized across the network, while on the other hand there is the network 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 deterministically 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.

## 3.2 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.

### 3.2.1 Interactivity: network instructions

Suppose now that we will allow a player to affect the game state, which is hardly a deterministic endeavour. 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[2]. 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 later 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 adding the physical network response time 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.

### 3.2.2 Synchronization instructions

What happens if instructions arrive 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 4.1.4.

### 3.2.3 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

[2]Servers and clients are not completely indispensable. Some games employ *peer-to-peer* networking where no server is appointed. The client-server model provides a centralized manner of handling and validating instructions, which is why we choose this model.

are traditionally played by no more than around 12 players, and with the low per-player bandwidth requirement this remains acceptable.

## 3.3 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* manner which exposes a continually updated non-blocking instruction queue to the programmer who can therefore easily integrate it 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 protocol TCP/IP. UDP is another protocol commonly used in games. It is generally used for more fast-paced games because it achieves faster response times by sacrificing among other things the guarantee of delivery: packages are sent almost without overhead, but some of them may never arrive, and those that do may do so in any order. The guarantee of delivery is essential, and along with the lax latency requirements this shows beyond doubt that TCP/IP is a better choice than UDP for our purposes.

The previous section established a *client-server* model, along with the concept of *instructions*. We shall further introduce the *protocol* which is simply a collection of instructions to be used by server as well as clients. The protocol consists of all the instructions that can be issued while the game is running, which would in our case include e.g. ordering the movement of a particular unit towards a particular location, ordering a unit to fire at a particular location, or the previously mentioned synch instructions.

Now we are in a position to propose the final layout of the networking package.

- `IOHandler`. Responsible for sending and receiving a particular type of instruction (for example movement instructions). An `IOHandler` has a `write` method, which writes the instruction-specific data (this could be a new movement destination for a unit along with that unit's identity) to the server. It has an `echo` routine which is invoked on the server when that server receives the information, such that the server may check whether the instruction is valid, thus preventing certain cheats. The server will then most likely just pass the instruction on to all clients after attaching an execution time stamp. Finally the `IOHandler` has a `read` routine which will be invoked when the client receives the information echoed by the server. The framework will provide input and output streams which the `IOHandler` can use in its methods.

- `Protocol`. This is an unmodifiable collection of `IOHandler`s which is identical across all computers, clients as well as server. In order to use an `IOHandler` it must be registered with a `Protocol` before connection is established. The protocol internally associates each `IOHandler` with a unique identifier which the client and server employ to distinguish types of instructions on the network.

- `Client`. The client can *connect* to a server at a specified IP address and port. The client will keep a thread running which listens for network input. Whenever input is received, the client will consult its protocol to alert the appropriate `IOHandler` to handle the instruction. Output to the server is written through the registered `IOHandler`s.

- `Server`. The server accepts connections from clients by listening on a particular port. Every client which connects will be registered, and the server will spawn a thread to listen for input from that client which terminates when the client leaves. Whenever input is received, the protocol is consulted and the appropriate `IOHandler` is made to handle the input. The `IOHandler` can then write any information it likes to all clients (it will most likely just pass on the instruction).

Finally there are *server-* and *client event handlers* which can be attached to the server and client respectively, which can execute code on connection, disconnection and *player events* (these are fired in the case a player changes name or team).

Using `IOHandler`s is quite easy: the `write`, `read` and `echo` methods must be implemented through subclassing. The framework will automatically pass references to relevant input and output streams as parameters to these methods (for example the `read` method is always provided with an output stream which writes information directly to the server, and the `read` method is provided with the input stream which reads data received from the server), which means the implementation only has to decide which data to write to them.

### 3.3.1 Implementation notes

The binary format used to send instructions consists of two parts, namely a header and a body. The body consists of the information which an `IOHandler` writes explicitly, while the header is managed automatically. There are two different headers, depending on whether the information is travelling from a client to the server or opposite. In both cases it is necessary to send the *identifier* of the `IOHandler` which is responsible for the instruction, such that the correct `IOHandler` can be fetched to handle the instruction at the destination. This information is currently written as a byte, though it has become clear that bandwidth is of such little significance that a 32-bit integer might as well be used.

When the instruction travels from the server to the client, an execution-time stamp must be supplied as well such that the clients know how long to enqueue the instruction in order to execute it at the same time as the other clients. The server will determine this timestamp based on a timer. Specifically the time stamp is equal to the current time, which the server reads from a timer, plus the server *latency* (mentioned in Section 3.2.1) which can be set when the server is created and adjusted at any later time. The time stamp is written as a 32-bit integer. Thus the instruction overhead is a few bytes, plus the overhead induced by the underlying TCP/IP protocol. The relatively small amount of traffic necessary to run the game renders this overhead unimportant.

### 3.3.2 Random numbers and deterministic code

Keeping games synchronized requires some care while running the game simulation. If the game uses (pseudo)random numbers, it is obvious that every client must be able to generate the same sequence of numbers, meaning that they should use the same random *seed* and that successive number generation should be deterministic based on the seed. The game world, to be discussed in Chapter 4, exposes a single random number generator which must be used *only* for events that are guaranteed to take place on all clients. Examples of operations that differ between clients are rendering. First of all the rendering rate is not fixed like the logical update rate, so random graphical effects should obviously use a local random number generator instead.

It is easy to transfer a random seed across the network at the beginning of the game (the client- and server event handlers are designed for exactly such purposes), such that all clients can use the same seed, but at this point all random seeds are still fixed to default values in order to ensure better reproducability in the event of bugs.

It is also obvious that non-deterministic mechanisms such as rendering routines should not invoke any method that can affect the game state. In our experience it is not difficult to distinguish between code which is deterministic and code which is not. During the development of JWars, we can proudly announce that we have on no occasion observed a game go out of synch unexpectedly.

# Chapter 4

# World of JWars

The JWars *world* is the entity responsible for handling the game logic at the highest level. The world encompasses coordinate space management, collision detector, vision model, event scheduling system and many other things which will be the subjects of this and several subsequent chapters. In this chapter we will describe some of the most fundamental properties such game flow and coordinate system management.

## 4.1 Flow of control and timing

Section 2.2 described how most games used a game loop, and went on to briefly describe the logical and graphical update mechanisms. Recall that JWars runs by means of a timer which performs updates with regular intervals, as opposed to an actual game loop. This section explains in greater detail why this timer is beneficial and what it does.

Note that game is designed such that the selected update rate has minimal impact on the game model. If a different update rate is specified (this is not yet possible at runtime, but a planned feature), units will still be seen to move at the same speed across the map, have the same reload time and so on, since the game generally specifies time intervals in seconds and converts this to frame counts internally.

### 4.1.1 Game loop vs. timer

A game loop serves to perform game updates at the fastest possible rate. For every update, units are moved slightly, e.g. by incrementing their positions by the movement speed times how much time elapsed since last update. Also the graphical display is updated, showing the new locations of units. If the update rate is high, animations will appear smooth and beautiful whereas lower update rates can make the game resemble a “slideshow” with movement occuring in large chunks. As game complexity grows, an update will take longer time for the computer, so units should move farther per update proportionally to the time which has elapsed. Using such a *variable update rate* ensures the best possible use of CPU resources.

As we have seen in Chapter 3, every client must conduct *exactly* the same logical updates, which forces us to use a fixed update rate instead of a variable one where entities would move based on the local machine's computing power. We have selected a frequency of 50 Hz, since this is sufficient for reasonably smooth animations while not too demanding for slower computers. Recall that only the game logic needs to be updated with this rate; graphical updates can be skipped if the local computer has trouble keeping up with the fixed update rate, resulting in less appealing graphics but preserving game integrity.

### 4.1.2 The game timer

The timer provided with JWars is designed to synchronize the update rate on different clients by periodically notifying *timer listeners*. It thus has similarities with the timers provided with the java core classes, but in fact provides more flexibility.

The listeners receive information about whether the timer is behind schedule (e.g. if updates are taking too long) such that they can decide to skip unnecessary operations. It also explicitly supplies the *frame count* which is obviously important to the simulation. After being *started*, the timer can be either *paused* or *suspended*. If the timer either paused or suspended, it will no longer perform updates until it is *resumed*. If it is suspended, then after being resumed it will try to regain the time in which it has been suspended by executing updates at the maximum possible rate. This is useful if temporary network trouble requires the game to halt temporarily.

The timer works on top of a *watch*. A watch is any entity which can provide the current time in milliseconds elapsed after some fixed point[1]. The timer polls the watch periodically and adjusts its update rate such that it never diverges from the watch.

The timer uses a thread internally. It is possible to specify that the updates should take place in the Swing *event dispatch thread* (see Section 9.1). In this case the timer will wait for the other thread to complete its update before requesting more updates. In other words the timer thread *blocks* until updates have been completed; another possibility is to enqueue multiple events after another if updating still takes place (the Swing timer will do this), but this does not provide the same flexibility: the former approach allows the next update to take into account whether too much time was spent during the last update.

[1]The watch should generally be some kind of wrapper around the computer's system clock. The java core API provides `System.currentTimeMillis` and `System.nanoTime`, where the latter is more precise but only available in newer versions. Programmers can choose the watch implementation freely, including third party timers.

### 4.1.3 Game performance discussion

During testing, we have observed that slow computers can have trouble keeping up with the required update rate during large battles where many units are moving. This means the computer will stop refreshing the display (except for sparse updates included to prevent the player from thinking that the game has crashed), and the client will gradually lag farther and farther behind schedule. Any received network instructions will then be enqueued for a very long time before the client eventually reaches their actual execution time, meaning the player will barely be able to control his forces.

This problem cannot be completely avoided: there will always be a computer which is too slow. The problem can, however, be remedied by optimization. Many of the operations carried out in the primary game update do not need a temporal resolution of 50 Hz. The figure of 50 Hz was selected because it allows for smooth animation. Might it be possible to lower the logical update rate without sacrificing smoothness? As we shall see in the next section, yes.

### 4.1.4 Interpolation during rendering

Suppose the logical update rate is very low, perhaps one tenth of a second. A unit which is located at $\boldsymbol{r} = (x, y)$ will in the next frame be located at $\boldsymbol{r} + \mathrm{d}\boldsymbol{r} = (x + \mathrm{d}x, y + \mathrm{d}y)$, and the distance between these points is so large that the graphical representation is no longer smooth.

However it is possible to perform graphical updates with a larger frequency than logical updates, while *interpolating* between the previous and current position depending on how long time has elapsed between the last and the current frame. Thus, if five graphical updates are performed for each logical update, each successive graphical update can depict the entity at positions $r + \frac{1}{5}\mathrm{d}r$, $r + \frac{2}{5}\mathrm{d}r$ up to $r + \mathrm{d}r$, and then the animation will retain its smoothness even though the internal representation does not. We can thus use the *weighted average* between the previous position and the current position to derive new intermediate positions, and

This powerful trick can easily reduce the CPU requirement of the logical simulation by 80% of the current amount (e.g. if the logical update rate is reduced from 50 Hz to 10 Hz). But things get even better. The variable update rate discarded in the last section can be reintroduced in connection with rendering, meaning that no particular graphical update rate is needed, but instead the rate can be dynamically adjusted according to requirements.

At present this optimization has not yet been implemented, but it is planned for the near future.

## 4.2 Coordinate spaces

It is normal for a computer game to utilize numerous different coordinate systems to represent information to the player (e.g. the screen coordinate system), or to represent the game state internally. It is therefore desirable to provide a standardized notion of coordinate systems to be used in the game. This allows for code reuse and reduces the possibility of bugs during the numerous coordinate transformations which would, lacking a centralized concept of coordinate systems, have to be coded manually throughout the game.

The basic requirements of such a system for our purposes can loosely be formulated already:

1. Locations should be represented by cartesian pairs of numbers (i.e. only two-dimensional systems are considered).

2. There must be a way to convert coordinates from any coordinate system to any other that represents the same space. This might involve scaling or other transformations.

3. There should be *tilemaps*, which we define as a coordinate system in which each location specified by a pair of integers is associated with exactly one *tile* object, the type of which can vary depending on the circumstances. Tilemaps thus resemble two-dimensional arrays which support coordinate transformations.

The notion of tilemaps deserves some comments. While real coordinate systems in mathematics tend to be *continuous*, i.e. have infinitely many points between any two different points, sometimes we desire purposely discretized representations. A chess board can be represented by a $8 \times 8$ tilemap where each tile can hold a piece. Similarly we shall find numerous uses for tilemaps in JWARS, including terrain representation and collision detection.

### 4.2.1 Coordinate data representation

While it would be nice to represent the world in continuous coordinates, this is obviously not possible using a computer. We shall have to select a way to discretize the world into some finite number of chunks.

Coordinate systems in games could conveivably be implemented in one of two distinct ways, representing positions either by floating point numbers or integers. Using floating point coordinates generally ensures a higher precision when calculating movement of units, while on the negative side it can be difficult to determine how numerically large coordinates may be before the floating point system loses precision. This can become a problem on very large maps. More importantly, floating point coordinates can be awkward in implementations where tiles are used, since tiles are naturally indexed by integers.

Since – as mentioned previously – we shall use systems of tiles for several purposes, which can only be indexed logically by integers, we decide to use integers as the basic datatype of world coordinates.

A coordinate system must be assigned a *width* and a *height*, which denote the number of units across horizontally and vertically, respectively. We shall refer to the number width×height as the *resolution* of the system. Two coordinate systems must have the same width:height ratio in order to represent the

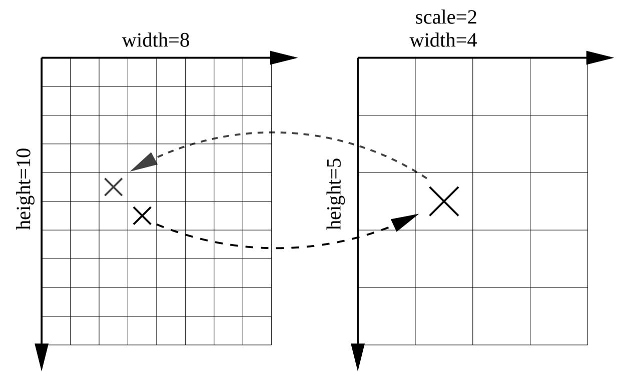


**Figure 4.1:** *Two coordinate systems. Axes are similar to those normally used with screen coordinates. The black × in the left system is transformed to the × in the right system, but the inverse transformation yields the grey × in the left system because of integer division.*

same space. Figure 4.1 shows two coordinate systems related to each other by a scaling of 2. A coordinate in the fine system is transformed to the coarse one by integer division, meaning several points in the fine system correspond to the same location in the coarse one (like many pixels could be part of the same *terrain tile*), whereas the inverse transformation is simply a multiplication. Note that we have decided to use the same axes with which pixels are normally indexed on the screen, but this choice is somewhat arbitrary. Presently coordinate systems only use scaling transformations with positive scale, but if the need arises they can easily be extended to use offsets or flip the axes.

Note to programmers: if a coordinate system's width or height is not divisible by the scale of a more coarse system (for example width=9 and scale=2), the coarse system is extended to include a slightly larger area than the fine system (corresponding to width=10). Under normal use this will not be a problem since all well-defined locations in both systems can be transformed back and forth safely, but carelessness can still lead to out-of-bounds errors.

The drawback of using integers instead of floating points is that movement must occur in chunks. If, for example, a game runs with 50 updates per second (which happens to be the current framerate in JWars), there is no intermediate step between a speed of 0 and a speed of 1 unit per frame, resulting in a quantization of speeds which can produce odd effects in the simulation. It would surely be awkward to have a speed of 50 pixels per second as a minimum.

Eliminating this problem requires a very large resolution of the primary coordinate system, such that the range of possible movement speeds seems continuous. For example, suppose the main coordinate system has a resolution of $2^{21} \times 2^{21}$, which means the map measures around two million discrete points across. If there are $2^9 = 512$ of these units for each pixel on the main display, and the game runs with a 50 Hz framerate, then the minimum possible non-zero speed is $\frac{1}{10}$ pixel per second, which is slow enough to depict a realistic-looking physical simulation.

### 4.2.2 Important coordinate systems

Many of the following chapters will introduce new coordinate systems for one purpose or other. We list here some of the coordinate systems that have

1. Main coordinate system. This coordinate system contains the logical coordinates of every entity and must have very high resolution.

2. Pixel coordinates. This is used for the representation of entities on the screen. For example an entity might be 20 pixels large, corresponding to several hundred units in the main coordinate system.

3. Terrain map. This tiled map contains large square chunks of terrain graphics used in rendering. Typically each such tile would have a side length of around 30 pixels.

4. Minimap. Most realtime strategy games use a *minimap* to represent a general overview of the situation, see Section 1.1.2.

There could be several other such maps, for example a coarse strategic map which evaluates the force strengths in regions for use by the AI or scoring system.

## 4.3 Terrain

Terrain representation is perhaps the most obvious application of tilemaps. JWars uses two layers of terrain: there is a basic terrain type (such as grass, which happens to be the *only* such terrain type yet included) and a vegetation layer. The terrain is represented as *terrain tiles*, where each tile must implement a *draw* method.

Terrain tiles also possess an index of *forestation*, i.e. the density of tree growth in the tile, which presently affects the movement speed of units and increases the toughness of infantry in those tiles.

JWars offers the possibility of adding objects that are not organized into tiles, such as buildings. These objects are referred to as terrain objects and will be described later.

Finally the random terrain generator allows creation of continuous 2D surfaces such as height maps by using the *diamond-square algorithm*. This algorithm was used to disperse trees throughout the world but could be used to generate hills and other kinds of terrain.

### 4.3.1 Terrain in games

Having different types of terrain in an RTS game is important for tactics. Terrain usually offers place for concealment and maybe even provide cover for units. In real life combat tanks are the masters of open terrain due to their long firing range and heavy armour combined the lack of cover for their targets. A tank in a city or a forest however does not have the visibility to keep enemies at a distance. A single soldier with the right weapon can disable a tank – if he gets close enough. To get close to a tank however a single soldier will need cover in the form of a terrain as forests, trenches or buildings. This is a single example on the impact terrain has on realistic simulations. A game constructed purely by the developers mind doesn't have to be realistic and can be constructed to not penalize terrain, but the JWARS game engine however uses realistic calculations and statistics, so terrain will be needed for balancing. Otherwise we could simply balance out tanks by giving all infantry squads a futuristic weapon which would counter tanks.

Terrain usually takes on two forms. The *natural terrain* on the battlefield like forests, hills and rivers and the more *special terrain features* in the form of objects like buildings, entrenchments or maybe even abstrabt objective markers. This system should be seen as a layer system. First is the ground features and form like hills and trees, on top of this comes the buildings and units. Each layer will be generated seperately so the data will be divided as well.

The terrain can have several impacts on the gameplay. Hills can have the effect of blocking *line of sight* concatenated to *LOS* as well as slowing down movements for units going up hill. A standard military strategy is also to have higher ground in both ranged as well as melee combat as effectiveness degrades when your target is on the high ground. Also forests can be implemented to affect gameplay through the loss of visibility and cover. A normal soldier in a forest would have plenty of ways to protect his body by standing behind a tree or lying behind a root sticking up from the ground. All these examples need some kind of terrain data representation for being implemented, and they can all have huge effects on the gameplay and tactics.

### 4.3.2 Map design

When creating terrain in any game there are two distinct ways of handling the map making which can either be combined or work independently.

**Pre-made maps.** Manually create all maps on which to play – possibly include a terrain editing tool for this purpose.

**Generated maps.** Implement a terrain generator which creates a unique map for each game.

Use of predefined maps can be necessary if it is critical that no player has terrain advantages. Few RTS games feature randomly generated terrain due to this obvious discrepancy concerning the game setup which makes the game unfit for tournements and other organized events.

In order for maps to be valid for tournement play they will have to be fair for all players in all starting locations. Maps created randomly will never offer the players equal opportunities or the same strategic options as a handmade map with a creative designers touch. With a selection of maps to use, many players also come to favorise some maps which they excel in - these features will be stripped by using random generators. Many games today rely on designed maps over generated terrain. Most of these however also include a map editor which gives players a chance to create maps themselves.

However there are positive things to be said of generated maps as well. Pre-defined maps will tend to foster pre-defined strategies. The strength of random terrain generation is that players will have to adapt to the circumstances, improvise and devise new tactics for every battle.

In a game like JWars both options are viable. In this project creating maps however is not high on the priority list, and creating maps manually would require a vast amount of time which could be spent elsewhere on the project. We therefore decided to develop a tool which could be used for random terrain generation using a particular seed[2]. The focus on this tool is not to create a complete map generator which will give a total map solution but merely an assistant for creating terrain features such as forests and height maps.

### 4.3.3 Random terrain generator

What we need is a way to displace height values on a two-dimensional grid in a somewhat random order which looks “natural” and not e.g. entirely random such as noise. We shall use the term *height displacement map* to refer to such a map. In real life, a forest will usually be a cluster randomly dispersed around a center – *natural displacement* – the same goes for hills. A height displacement can be obtained by using *mid-point displacement.* The mid-point displacement method works by starting with a straight line, the end points of which are at the same height, see Figure 4.2. The middle point between the two end points is then displaced by a random number, for example between 0 and 1. There are now three equally spaced points forming two line segments, and the next step consists of again displacing the points that are at the middle of each line segment by new, limited random amounts. Continuing this, we get first 3 points, then 5, 9, 17, 33 and so on. In general if we stop after $n$ steps, the line will be divided into $2^n+1$ points. As figure 4.2 shows, this method ensures a continuous landscape curves instead of random jaggy spikes.

In theory what is needed is a two-dimensional surface where each point has a height. This third dimension can be interpreted to be height displacement (obviously), vegetation density and any other concept which can be characterized a function $\mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$. Our present problem it is not immediately obvious how the mid-point displacement method can be used to generate a *two*-dimensional grid.

[2]Using a seed for creating terrain allows us to create random maps but still save promising seeds which can be used to reconstruct those maps

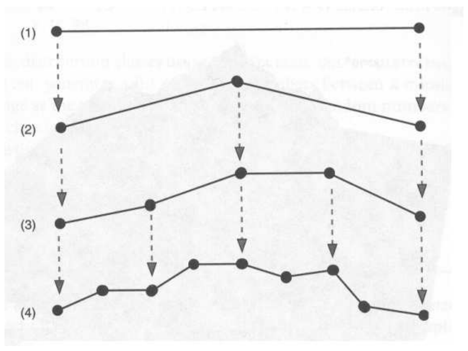


**Figure 4.2:** *The mid-line displacement used on a one-dimensional map*

One algorithm which applies mid-displacement to generate a two-dimensional grid is the *diamond-square* algorithm, which we have chosen to implement. The algorithm, shown in work on Figure 4.3, is again based on steps - starting with the large scale displacements and then with successive iterations creating smaller displacements. The algorithm works by first displacing the *corners* of the grid, corresponding to the end points in the one-dimensional case, and the center, to random initial values. Then the four points at the centres of the map *edges* are introduced, by assigning a height value equal to the average height of the nearest *neighbouring* points that already exist, plus a small random amount. The edge mid points actually have four neighbours if we consider the map *wrapping*, i.e. the leftmost points are neighbours of the rightmost points (thus theoretically resulting in a toroidal topology).

The algorithm earns its name because it successively displaces points which are located on a grid (squares) and in diamond shapes. Implementation-wise we start with two-dimensional grid of floating point numbers, where the grid sides must have a length of one plus a power of two. Now, the algorithm consists of the following steps, illustrated on Figure 4.3 (*traversing* in this case always involves setting the affected points' heights to the average of the neighbours plus a small random amount as mentioned before):

1. Set a *step* value equal to the width of the map.
2. Traverse the map using steps of that size (this results in the corners being traversed on the first run, as in Figure 4.3a) and the central point on the first iteration (Figure 4.3b.
3. Traverse the map in a *diamond* pattern using the same step size, Figure 4.3c.
4. Divide step size by two, and increment the origin of the grid by a quarter of the step size for the next iteration (this means traversals otherwise starting in (0,0) will now start in (step/4, step/4). Go to step 2 unless all points have been traversed.

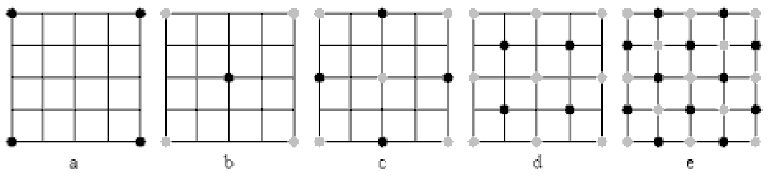


**Figure 4.3:** *The diamond square algorithm running on a 5x5 grid until termination. If the grid were larger, the first steps would be identical, but the algorithm would continue by halving the* step *size and performing the same operation over and over.*

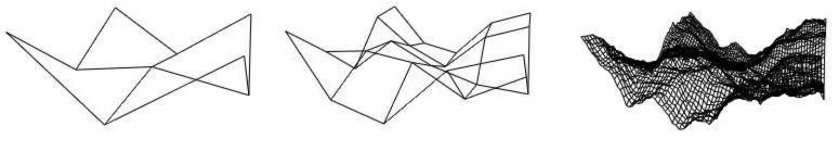


**Figure 4.4:** *This is a 3D model of a diamond-square algorithm running on a NxN map. It clearly illustrates how jagged maps come to look more natural.*

This method eventually gets to traverse the *entire* grid. Figure 4.3d and e show the second iteration. An example 3D model of a generated map is seen on Figure 4.4. Two-dimensional map representations generated by our implementation are seen in Figure 4.5.

In the current version of JWars, the growth of trees is deciding by interpreting a generated "height" map as a tree density.

In the terrain generator uses a class is called a `FloatBuffer` to store the two-dimensional grid. It wraps array of floats and it is on this array that we perform the diamond-square algorithm. The float buffer has several functions that can be used to modify generated maps:

1. `cutOff` : takes a float as argument. All elements in the array get the float value subtracted from them. If the new value is below zero we round it off to zero. This is used for creating the forests – at many locations the forest density should be exactly 0, and this is achieved by flattening the map with `cutOff`.

2. `smoothify` : evens out the terrain more by traversing all non-boundary elements and setting them to the average of the elements neighbours. We use this function several times on the map used for creating the forestation levels.

3. `scaleToFit` : Takes a minimum and a maximum value as arguments. Scales and translates all the values in the buffer by the same amounts,

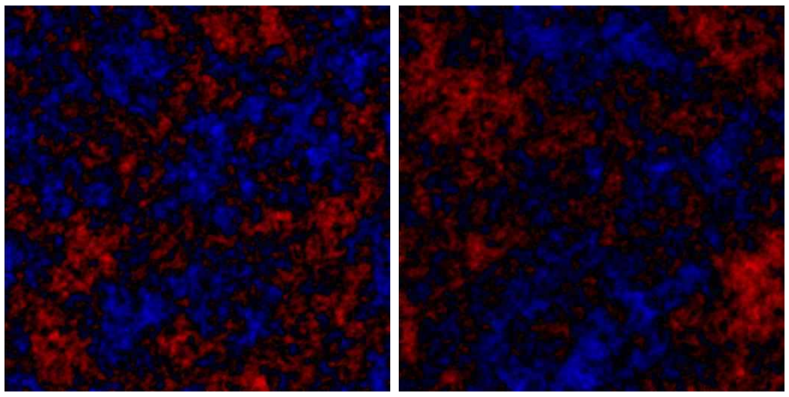

(a) (b)

**Figure 4.5:** *These are two maps randomly generated by the terrain generator. It is not immediately noticable but both maps are periodic, i.e. their edges wrap.*

such that the smallest and largest values are as specified. This method guarantees that the float buffer does not contain elements larger or smaller than the maximum or minimum.

The height generator algorithm and the float buffer constitute a powerful tool when used together.

### 4.3.4 Terrain objects

After having all the natural terrain have created we can add the second layer of terrain features to the world. These are called *terrain objects*. Terrain objects can take the form of fortifications, buildings impassable mountain peeks, roads or even lakes.

For creating terrain objects we impemented the *blueprint* class. Blueprints are used in JWARS as blueprints in the building industry. A blueprint contains all technical data concerning a specific objects, and when you have a blueprint you can make a building as specified by the blueprint anywhere - even build several buildings based on the same blueprint. The blueprint contains all relevant data for an object except the location and an angle. At this moment all blueprints and terrain objects are created in the *StructureFactory* class which contains methods for generating objects and disperse them on the battlefield.

For creating a blueprint it will need a shape and some collision properties. The shape is created by consisting of coordinates in an array, where each coordinate designates a corner on the object, while the collision properties are handled like on an unit – it can be massive or not. For upholding the ideals of JWARS the blueprint can have any shape and size. The coordinates are placed in an abstract coordinate system centered on the blueprints center (0,0). Along with the coordinate array the blueprint creates the *collision properties* according to the developers wish. For creating a terrain object we feed the constructor with an image[3], a blueprint, an angle, and the location within the world. After the terrain object has been created we can add it to the world by registrering it in the detector.

For making buildings usable by the pathfinder we create a *pathfinding node*, and attach it to each corner of the object. The pathfinding nodes are only created for the pathfinder usage. All polygon objects will be treated as convex hulls by the pathfinder even though they might be in the shape as a hourglass.

### 4.3.5 Terrain appereance

I this section we will discuss the appearance of terrain JWARS. The appearance of terrain randomly generated. The terrain consists of *grass*, which serves as the *ground*, and each tile has a *forestation level* which designates the amount of trees in that tile. Each terrain tile contains references to two images – one for the ground and one for the vegetation. Most of the images are shared by multiple tiles. For example there are sixteen different images of grass to be shared by hundreds, maybe thousands of tiles on a map. The sixteen variations are used to make the terrain look less monotonous. For each *forestation* level, six different images are created for this purpose.

Both grass and forest images are referenced within each terrain tile, being used by the tile's `draw` method whenever the tile should be rendered to the screen. The tile thus knows how to draw itself, and it would therefore be possible to use terrain tile implementations that do *not* rely on images along with those that do.

Creating the terrain map, along with the generic grass background and the different amounts of forestation is done in the `MapFactory` class. Specifically the map factory distributes the tiles, using different variations of the images, randomly in a terrain tile double array. Figure 4.6 shows how hard it is to see any repetitive pattern in the background. When examining the background the same image can be spotted several times on the screen but it will not annoy the player or make the background layer seem "too" generated.

We have found a special combination of colours with an added amount of randomness yield a satisfying result - each pixel in a grass image has the following profile in RGB – the RGB values are (18,96,6) plus random numbers up to (128,72,72). The grass images thus contain only *noise*, and therefore seem to fit beside each other.

For creating the forest graphics we use the same principle. Instead of creating a single array as with the background images we need several different images for the different amount of forestation. The default amount of levels in forestation is 6 and we have chosen to created 6 different images of each level. By moving the RGB scale to the darker greenish area small circles representing trees are

[3]Images can be attached to objects but at this moment in development we simply fill polygons with a colour instead of making images for all buildings.

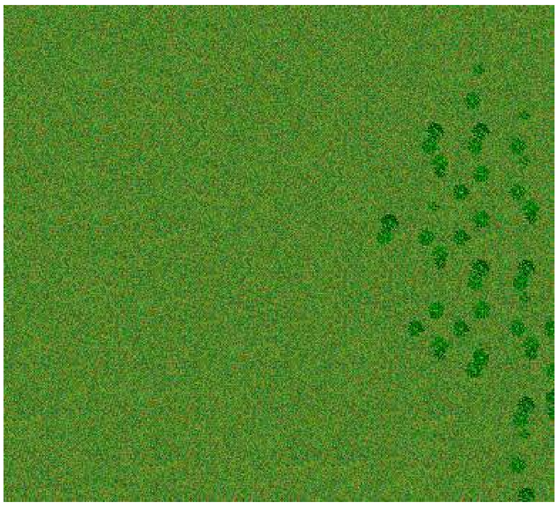


**Figure 4.6:** *Terrain graphics. Though the terrain consists of tiles, 32 pixels on each side, this is not clearly visible due to the amount of varying tile images. Some trees are visible in the right side of the picture.*

painted onto the image. By first calling a terrain tiles draw function and then the drawVegetation we get all the layers in place.

The technique described here could be expanded to support different terrain types by using the appropiate RGB codes for each wanted graphic set. This is however of minor importance for the development of the project and is still a feature designated for future implementation.

One of the original thoughts was to let the terrain generator create a height map and include it in game engine. This would have made the game engine more realistic but would also rise a new problem: how do we illustrate it? Height curves are not implemented in the current version of JWars, but they are planned for the future. Using 2D graphics make illustrating a third dimension in a map somewhat difficult. The solution has been around for some time in real-life map making where height curves illustrating terrain differences would be a viable, but ugly solution.

## 4.4 Event handling

Many if not most real-time games include a *game loop*, which is a loop in which the entire model and graphical display of the game are updated repeatedly. This normally involves traversing all the dynamical entities and updating their positions, velocities and other variables. These updates might include operations such as the creation or removal of entities from the game, which can be inconvenient while the list of entities is being traversed. It is therefore desirable to handle updates in one loop, then store the more complicated operations as *events* to be resolved later, just after the game state has been updated. This approach can prevent bugs and ensure that things are done in a consistent order.

Fundamentally we shall here refer to an *event* as something which can be put in a queue and then *executed* at some later time. Note that in this model, the event serves simply as enqueueable executable code, which is in contrast with the AWT/Swing event term, where events are short-lived objects that convey specific information to event listeners.

### 4.4.1 Types of events

There are three distinct event concepts which will prove useful.

- Peripheral input. The user can typically control the game by mouse, keyboard or typing commands into a console. It can prove troublesome to invoke the code associated with these actions immediately: if the player e.g. changes the view of the battlefield *while* the battlefield is being drawn, this will result in graphical tearing. This should not happen, and this kind of event should therefore be stored and the corresponding code executed only when graphical and logical update operations have been finished.
- Network events. As we shall see in Chapter 3, instructions received from the network are *scheduled* to be performed at specific times. Therefore these instructions should be enqueued until that time.
- Delayed events. If weapons are firing, then their reload progress must be tracked somehow. This could be done by polling *each and every single weapon* (of which there are probably hundreds) once per update, but if they reload equally quickly then it is simpler and more efficient to insert *reload events* into a queue such that it is sufficient to poll that queue of events once per update.

### 4.4.2 Performance considerations

While the storing of multiple events in the same queue (like in the reloading example above) can eliminate most of the checks otherwise necessary, there will still be an abundance of events to be allocated in memory and released. It is therefore desirable to save some of the frequently used events such that they can be used multiple times. Following the earlier example with weapons reloading, it would be expensive to create a new reload event every time a weapon fires. It would be more sensible to save the old reload event and enqueue it again the next time that weapon fires, because the weapon obviously cannot fire before its reload event is released from its queue.

### 4.4.3 Queueing system

The preceding discussion leaves us with two primary concerns, namely an *event* and a *queue* which can store events. The event should have an *execute* routine and it should know the time at which it is supposed to be executed.

The queue should have an *update* routine which polls the next event in the queue for whether it should be executed, then executes it (and possibly any following events) if the time is right.

This is enough to handle the *delayed* and *network-type* events as noted before. In the example regarding reload of weapons, it will be necessary to use one queue for each different reload interval. For example, if rifles can shoot once every 100 frames then all rifle reload events can be stored in a rifle reload queue, and all grenade launcher reload events can be stored in another queue representing another reload time.

Finally, peripheral input events should generally be handled immediately (i.e. within the same update as it is generated), but this kind of input could originate from another thread than that in which the game updates are performed. It is therefore necessary commendable to use a thread-safe approach (in java this is done simply by declaring the relevant methods `synchronized`).

In conclusion we now have two special queues, namely the peripheral input (synchronized) queue which executes the events stored in them immediately when polled, networking queue which stores instructions received from the network until such time as they should be executed, and any number of delayed-execution queues that handle weapon reloads and other things which we shall see in other chapters, such as vision checks and targetting.

# Chapter 5

# Collision detection

This chapter will after an introduction to collision detection formulate the design and capabilities of the JWars collision detector. It is designed to handle large numbers of geometrically simple colliding entities without constraints on entity sizes.

## 5.1 Basics of 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. We shall 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, possibly 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.

### 5.1.1 Divide and conquer approach

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 not the case in a normal real-time strategy game.

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 (see [2, pp. 28-33]), though it is not explicitly recursive.

The best case scenario where all units are in different tiles runs in $\mathcal{O}(n)$ time since no cross-checking takes place. The worst case scenario, where all units are in *the same* tile is extremely unlikely, because only a handful of units should fit physically into a tile.

### 5.1.2 Tile registration strategy

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. 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 sufficient.

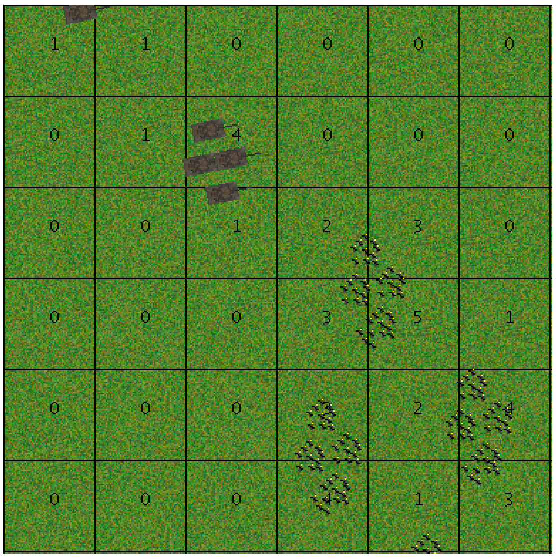


**Figure 5.1:** *The collision grid visualized. The number of units registered in each tile is listed inside the tile. This is an in-game screenshot; the debug grid can be enabled by passing -d as a runtime parameter.*

For the JWars collision detector we have chosen the second approach, which is illustrated on Figure 5.1.2, 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 tiles would be a bad case in this model whereas the former model consistently requires checking 9 tiles). However there is one possible problem, namely that two units which occupy two of the same tiles will (unless carefully optimized out) be checked against each other in both of those tiles[1].

### 5.1.3 Shapes and sizes of colliding entities

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

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

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 the realtime strategy genre units are commonly modelled as circular or square, since a larger degree of detail would hardly be noticable on the relevant scale. We have therefore decided to provide only collision detection for circular units. However the 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 is sufficient for most basic entities. However there are presently static objects (see Section 4.3.4 on terrain objects) which are polygonal and can be very large, and they make use of this escape mechanism.

## 5.2 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). Whether or not a collision has been detected is determined solely by these properties.

### 5.2.1 The checking routine

The entire checking routine for a single collider which wishes to move to a certain location now reads:

1. Determine which tiles the collider will overlap in its new position
2. Traverse these tiles, and for each other collider found here, perform the following steps.
   (a) Check whether the bounding circle of the moving collider intersects the bounding circle of the other collider.
   (b) If the circles intersect, invoke user-defined checking routine.
   (c) If the shapes intersect, invoke user-defined collision handling routine on the moving unit. The moving collider will not be moved to its desired position, and the checking routine is terminated.
3. If at no point above the checking routine has been terminated, the moving collider will have its position updated to its desired location. The collision tiles overlapped by the collider in question will be updated accordingly.

This routine works well in the realtime strategy genre when the primary function of collision detection is to prevent entities from overlapping. There is no particular way of *handling* a collision other than cancelling the movement request (unless the user specifies this manually in the handling routine), and this approach would therefore be bad if realistic physics (conservation of momentum or elastic collisions, for example) were desired. These things are not particularly relevant in the realtime strategy genre where the behaviour of a single unit is not closely monitored.

### 5.2.2 The collision grid

In order to represent the collision grid, the collision detector uses the *map* utility package which is described in section 4.2. It fundamentally 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 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.

### 5.2.3 Further features

Finally a few utilities of the collision detector should be mentioned.

First, 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* (a "2.5 D" approach where a two-dimensional world is artificially equipped with a few layers representing different heights) 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.

There is one more function that can advantageously be included with the collision detector, even though it does not relate directly to collision detection: Section 9.3 describes how entities are rendered to the main JWars display. In order to localize the entities that are actually present on the display, it is desirable to traverse the tiles used by the collision detector. The collision detector should therefore also have access to the terrain map. When an entity is moved, the collision detector is in this context responsible for *dirtifying* the affected terrain tiles, meaning that those tiles should be redrawn during next graphical update. This process, traversing the overlapped terrain tiles, is completely equivalent to that of traversing collision tiles. With this in mind, each collider must also possess a *sprite*, the concept of which is described later in Section 9.3. The collision detector thus tracks the movement of sprites on the screen, such that redrawing can be skipped in regions where no movement takes place.

### 5.2.4 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 (see Section 4.1.4) and perform interpolation to ensure graphical smoothness between logic updates (thus using a higher graphical than logical update rate). Using such an approach the performance could be enhanced 10-fold, and would allow the collision detector to handle at least $10,000$ moving entities on our test system, but this figure can be reduced if custom geometries are used or if other parts of the logic are computationally heavy.

### 5.2.5 Using the collision detector

The programmatical interface of the collision detector is very simple and can be concisely described in only few terms:

- The collision detector is instantiated by supplying three coordinate systems, namely the high-resolution *main* coordinate system of Section 4.2, a tile map of *collision tiles* and a terrain map (Section 4.3).

- An entity, technically anything which implements the `Collider` interface, can be added by calling the `register` method, passing a reference to the collider in question as parameter.
- If an entity is to be moved, the `move` method should be called, specifying the relevant entity and its proposed new location. This method will, as described above, check the validity of the new location for the entity and move the entity accordingly. If a collision is detected, collision handling methods on the colliders in question will be invoked as required. Finally this method returns whether the move was successful.
- An entity can be removed from the collision detector by calling the `remove` method.

If for some reason the locations of entities are changed *without* notifying the collision detector, this may result in that entity being registered in incorrect tiles. Thus that unit might overlap other units without a collision being reported. This issue can be remedied by covertly encapsulating the positions of entities within the collision property such that it is impossible to tinker with it from outside; at present we have not deemed this precaution necessary.

## 5.3 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 the entities, meaning that all units are considered circular. However an escape method is provided that allows arbitrary geometry.

Performance-wise the collision detector is optimized for large amounts of units each with simple geometry, but even if complex geometries are used the combined use of bounding boxes and bounding circles is likely to eliminate most of the expensive checks.

# Chapter 6

# Pathfinding

The JWars pathfinder is a modification of the well-known A* algorithm, which is specialized to handle large and open maps.

Pathfinding is an essential part of any real-time strategy game as it enables the player to control units without wondering if they make it to the selected destination or not. Requiring the player to find the suited paths for all his untis is out the question, as it would become infeasible for any human when the unit count reaches a large enough number. The best solution is to let the computer calculate a path for the unit through the world, which would satisfy the player.

The best way is not necessarily the fastest, since it can, for example, be more dangerous to walk on a road when enemies are nearby. It is probably better to select the geographically shortest, which may lead through rough terrain, but this behaviour is more predictable for the human player.

## 6.1 Pathfinding in general and in JWars

Moving units in RTS games require a pathfinding algorithm to navigate around impassable obstacles. Most game pathfinders today extend the normal 'single-source shortest path problem' solution to incorporate unit-to-unit relations, which make units capable of interacting in order to navigate around each other dynamically. For this project we need a pathfinder to work on the world of JWars, while it should still be a viable solution in other world representations. Given the world representation in JWars the pathfinder will likely be used on large maps, and with no restrictions on terrain objects shape and size: it will have to be very adaptive.

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.

Many contemporary RTS games solve the problem by using a tilesystem. When using a tilebased pathfinder the world is structured in to tiles and units, buildings or other entities take up space by having the ability to *occupy* tiles. The map used for pathfinding designates tiles with either “used” or “free” as markers when scanning through the map with an algoritm[1]. This approach has several advantages, like high and consistent speed, while it requires a map-structure supporting this to search in. As described in Section 4.3, we wish buildings and other terrain objects to have a certain amount of flexibility (for example, small buildings should not have their shape determined by an inflexible grid), thus having minimal restrictions on shape and size. We therefore choose to allow polygonal terrain objects, and thus relying solely on a tile structure to simplify the problem is no longer feasible.

For JWars a different approach must therefore be used. In order to devise an algorithm we look at the basic pathfinding problem – an object is blocking your path. The shortest way around an object is to walk around it, either left or right. Using this idea the algorithm should “walk” (or *shoot*) in a straight line from the starting point towards the destination point until it meets an obstacle. It will then examine the obstacle and generate paths *left* and *right* around the obstacle, then shoot again from each side of the obstacle. Note that when using polygonal objects, it will be optimal to walk along the obstacle’s line segments until reaching the corner from which the destination is again “visible”: corners can be used as intermediate waypoints. Eventually it should either reach its target or decide that the target is unreachable.

### 6.1.1 The algorithm

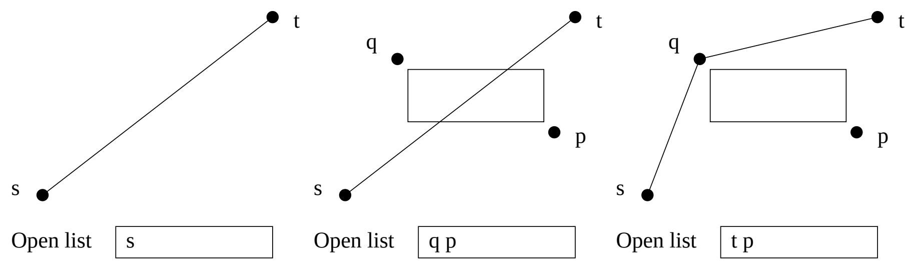
We will here as a general overview summarize the workings of the derived algorithm. More detailed, but difficult, observations will be postponed to the next sections, along with a full description.

While running, the algorithm maintains a list of *potential* waypoints (or *nodes*) called the *open list*, which works as a *priority queue*. The priority queue keeps track of the nodes immediately reachable by the algorithm and sorts them using a heuristic evaluation in order to estimate which way will most likely be the fastest; this will allow the algorithm to guess the correct way without having to try all combinations of left and right which could take a long time if it has to walk around many buildings. The example in Figure 6.1.1 shows the algorithm at work in a simple setup. The list of nodes is updated and sorted after each iteration in the algorithm, and nodes which have been accepted as waypoints are removed from the list (they are no longer *potential* waypoints).

Let us go through the steps taken by the algorithm:

1. See Figure 6.1(a). The algorithm is about to shoot from s to t. Having not yet started, the open list contains one element, being the starting point s.

[1]Although these are not open source games, meaning that we cannot know for sure, several observations support this assertion. For example, buildings can typically be placed only in discrete locations, and in some games units in close clusters (notably *zerglings* in *Starcraft*) are clearly placed according to a grid.



(a) About to start. (b) An obstacle blocks the way; either walk towards q or p. (c) q is slightly shorter.

**Figure 6.1:** *A simple pathfinding problem.*

2. On its way the algorithm discovers an obstacle. It determines the "leftmost" and "rightmost" points q and p as seen from the starting location. Having stepped onto the starting location, s is removed from the open list while p and q are now potential waypoints. The algorithm "guesses" that the distance s-q-t is smaller than s-p-t and therefore it sorts the open list with q before p, meaning that it will check the most promising path first. This is shown on Figure 6.1(b).

3. Now q is removed from the open list, as the algorithm shoots from that location. Note that p remains in the queue in case another obstacle is discovered which makes the current path longer than expected. However no other obstacle is found, and thus t is reached and added to the open list which now reads (p,t). The algorithm sorts the list, determining that the distance travelled (s-q-t) plus the remaning distance (0) is still smaller than s-p-t, then sorts the list which now reads (t,p), see Figure 6.1(c).

4. The algorithm finally terminates when t is removed from the open list.

This largely explains how our algorithm works. However there are still uncertainties, such as the exact strategy used to "guess" which distance is the shortest. This shall become clear in the next sections.

Logically this method favours sparsely populated areas since fewer objects would create fewer obstacles and result in more straight lines. A pathfinder based on a grid system (where the individual tiles serve as nodes) would have smaller search areas if lots of buildings are occupying space, thus leaving less free space to be searched through. Our pathfinder will have the opposite problem: in large, sparse areas there are few nodes, but a labyrinth would be a mess to represent because of the scores of corners: our chosen algorithm is specifically designed to represent large, outdoors areas.

### 6.1.2 Data structure

Most people proficient within the pathfinding area choose to run their algorithms on *graphs*. A good example of an algorithm using graphs is the A* algorithm which is a shortest path graph algorithm. For finding a shortest path using graphs for data representation, history has shown that the A* algorithm is a viable choice.

In any situation we will need a way to represent possible future waypoints of a moving object as fixed points so e.g. a move order can be broken down into multiple segments represented as a graph. Given a graph represented as follows:

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

$V$ is a list or other representation of all the *vertices* (or nodes) 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 $v2$ if they are connected by the edge $e(v1, v2)$. We shall also introduce the *weight* of an edge, corresponding to the amount of time (or the cost) it takes to traverse it, which is given by a weight function $w : E \mapsto [0, infinity]$.

Given a graph with a chosen data structure there are several possibilites 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, as these have the best running times with the fewest vertices visited – like the A* algorithm.

The pathfinding in JWars has some requirements to the algorithm which we must take into account before finally choosing a solution. The most pressing issue is to handle the dynamic and rather limitless implementation of units and other objects in the world (recall that the collision detector, Chapter 5, allows arbitrarily sized units and obstacles). We have chosen a very open approach which imposes only limited restrictions on unit and building location, size and geometry, which however complicates the final form of a pathfinding solution. Any building or unit can be placed anywhere on the almost continuos map and will thus not e.g. fill out a predefined amount of tiles in the world. In order to perform pathfinding we need access to the units and obstacles placed in the world. Therefore the most obvious data to use for pathfinding are the actual objects stored in the collision detector, Chapter 5.

If we are to use the object data some rules have to be defined, or the amount of different scenarios is limitless. An effective yet relatively simple way to this is, as mentioned in the preceding example, to represent objects as polygons. More specifically it proves necessary to allow only convex hulls. Convex hulls have many properties which make the basics of handling and calculating a lot easier. If we do not establish ground rules like this the more eccentric objects will be impossible to handle.

In this project it is the data representation and requirements for the world modelling which forces us away from the normal pathfinding implementations. For this game we will have to come up with a rather unique pathfinding solution.

As stated above the best data for these calculations are the terrain objects since they *alone* contain the relevant data. A solution to a pathfinder using only the terrain objects can be as simple as walk towards the goal, if you encounter an obstacle walk around it and continue towards the original goal. On this basis we have developed a pathfinder which is based on the A* algorithm and employs a heuristic estimation of the distance from any node to the goal. The JWars-pathfinder is meant for 2D purposes only and in this case a straight line towards the goal will result in the most optimistic evaluation a node can get.

## 6.2 Implementation

For using the pathfinder some unique classes have been implemented. The pathfinder is designed to work on objects of the class `TerrainObject`. All terrain objects have a list of *pathfinding nodes* which the pathfinder uses as vertices. In order to work on the pathfinding nodes using the A* algorithm, the vertex must possess several attributes. These attributes are as follows: a reference to the *ancestor* of the vertex (i.e. the previous node in the path) and three integer values which we call $f$,$g$ and $h$. The three integers are all measurements of distance. The variable $f$ holds the distance travelled during the algorithm to the current node. $g$ holds a heuristic evaluation (or guess) of the distance to the goal from the current vertex, and $h$ is the sum of $f$ and $g$. The attributes of the pathfinding node are essential for understanding the more technical description of the pathfinder.

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 with the following rules and definitions.

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

The start and goal locations are considered vertices[3] which are specified for each run of the algoritm.

### 6.2.1 Expanding and searching

The algorithm is started by calling the method `findPath` with an end coordinate and the unit for which a path should be found. As explained later the pathfinder returns unique solutions to specific settings. Calling the method with two differently sized units can yield two different results. This will be described to depth later in this chapter.

[2]If it were to be a pre-defined list for $E$ 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 a lot of memory space.

[3]The pathfinder contains a specific class for this purpose called `Target`. This class extends the the `PathFindingNode` class and can also be registrered in the collision detector.

Given the *start coordinates* as the unit's current location and the *end coordinates* as argument to the method, we can create the start vertex and add it to the priority queue. The pathfinder uses the standard loop from A*, which means it expands the search area from the first element of the priority queue; it will therefore be forced to select the start node for the first iteration. In a standard implementation of A* the priority queue will be referred to as the *open list*.

Taking into account that all distances travelled are straight lines, we can always be sure that we have the shortest possible path between any two given nodes if we use the "relax"-concept as in [2, p. 586] when describing Dijkstra's algorithm. A pathfinding node's $g$-score is simply calculated as 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 possible result of getting directly to the *end node* will be next in the priority queue. This approach mean we can safely terminate the algorithm upon reaching the goal location and have the shortest path possible without further extending the search area. For extending the search area we let the algorithm draw a line between two nodes and check the line for collisions. This is done using the *expand* function in the pathfinder. Having the loop selecting a new node to expand by each iteration we will now explain the *expand* function and how this works in the world of JWars. When expanding a node we seek activate nodes which can be reached in a straight line from the current node. We do not seek *all* possible nodes, merely those who will prove beneficial for further searching. When expanding a node we expand it *towards* another node - this being either the target node or the corner of an object. The expand function is used for expanding the search area of the algorithm as it adds newly discovered nodes to the priority queue. If the path between two nodes is not blocked by any object, we can safely add the target node to the priority queue, as we can guarantee a direct path between the two nodes exist. If an object is blocking the route between the two nodes, we try to find a way around the object by calling the expand recursively.

The `expand` method determines how to expand the search tree, by finding obstacles and recursively searching the paths left or right around them. Written in pseudocode, reads:

```
expand(source, destination, unit){
  [use Bresenham's algorithm]
  tileList = getTileList(source, destination)
  obstacleList = getObstacles(tileList)

  for each obstacle in obstacleList
  {
    if( path might intersect obstacle )
    {
      angle = angle from source to destination
      minAngle = angle from source to obstacle's leftmost corner
      maxAngle = angle from source to obstacle's rightmost corner
      if( minAngle < angle < maxAngle)
      {
        expand(source, leftmost corner of obstacle, unit)
        expand(source, rightmost corner of obstacle, unit)
      }
    }
  }
}
```

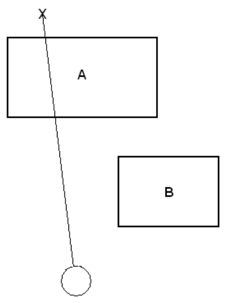
If we hit the wanted pathfindingnode while finding min and max values the node will be added to the priority queue and is then activated for future expansion according to the heuristic evaluation.

The recursive call to the expand function enables the function to activate several edges leaving one node thus activating all relevant edges for leaving the current node. A single node expanded could follow Figure 6.2

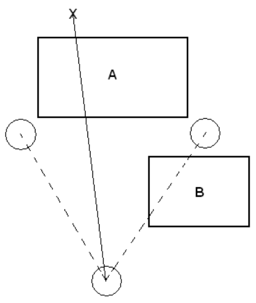
Every time a position (pathfindingnode) is grey, Figure 6.2, it has been added to the priority queue by the expand function. When the expand function succesfully makes contact with the targetted node we update the target node with the relevant data for the A* algorithm to run as intended. The update method will reevaluate the three values needed for sorting and evaluating nodes in the list so we can expand further according to the heuristic evaluation. Finally it will set the ancestor of the given node to the node from which we came. In theory no edges are represented in $E$. When a node is expanded we get a set of edges based on the current pathfinding problem. The expand function is essential for this pathfinder as it is the major difference between our pathfinder and a more convenitonal pathfinder with defined edges for each vertice.

In JWars the class `PathFindingNode` has been implemented solely for the purpose of pathfinding and has all the needed attributes for being handled as a vertice. A pathfinding nodes settings is calculated from the blueprint which determines the objects size, shape and positioning. A very important feature of a pathfindingnode is the ability have a static coordinate and a dynamic coordinate. This ability is neccessary for the pathfinder to find a path based on the `Moveable`'s radius. When creating a `PathFindingNode` a vector is calculated based on the two adjacent corners in the object creating an indent direction. When multyplying this indent direction with the unit radius we get an *indented location*. This location is the dynamic coordinate which will be calculated in each run through the pathfinder for all relevant nodes. In order to locate obstacles in a line between two points, the pathfinder uses a specially endowed tilemap called a `LineDrawCapableMap`.

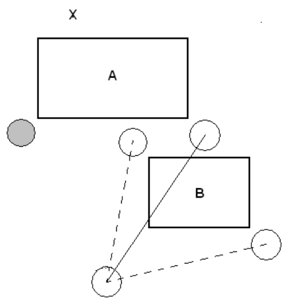
The `LineDrawCapableMap` comes with a method which utilises Bresenham's line drawing algorithm to find a list of tiles based between two points on the map. A LineDrawCapableMap can be constructed on top of an ordinary tilemap, providing the line drawing capability to a tilemap which originally could not offer this functionality. Specifically we want to endow the collision map (see Chapter 5) with the ability, since this is an obvious way of finding obstacles on



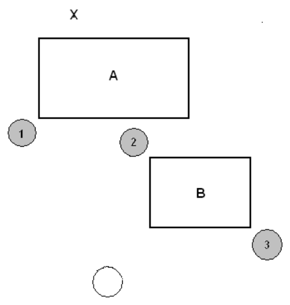
(a) By first expanding towards the target object A is found to block the path



(b) When expanding towards the corners of A, one call is succesfull and can add A's corner to the priority queue while the other finds object B to block



(c) With B blocking the searched route towards A's second corner we need to establish routes towards B's corners.



(d) Both expands towards Bs corners are successive and they are added to the priority queue.

**Figure 6.2:** *A single iteration in the loop of the pathfinder. The expand function calls it self repeatedly so all needed nodes are found.*

the path. From the LineDrawCapable map a list will be returned consisting of CollisionTile's from the collision map. The line drawn between the two points can be ordered in any thickness (measured in collision tiles) required for units larger than the standard collision tile. Using the list of collision tiles we have access to all registrered objects in the vacinity of the searched path.

When checking a building for collision we take several steps before concluding that a collision will occur. The free positioning and shape of objects makes a simpel point-to-line distance worth calculating. This will ensure that buildings with no chance of interfering with the searched path will be excluded from the check early on. The second step is to calculate all angles to the the indented locations in the current object. Calculating the largest and smallest angle we can perform a check wether the line is between these two angles. If we detect a collision with the object, we enforce the rule about all objects being convex hulls for pathfinding issues. If we need to go around the object we references to the largest and smallest angle to the object. Now we simply expand towards these nodes as stated in the pseudocode for the expand function.

By using the dynamic expand dunction we have a new setup and all nodes could produce a new set of edges everytime we use the pathfinder. We do not store the individual edges but merely activate those discovered by the algorithm upon expanding a node. Using this approach we expand the graph accordingly to the A* and update the nodes found by the expand function. [4] The operation that makes this algorithm stand out is the expand function which activates vertices/edges while searching for the path.

An important aspect of the chosen solution is that it is only dependant on the game implementation of the collision detector. If a developer wants to use this pathfinder it is fairly easy to convert to a different setup - a conversion would need a function capable of detecting a collision between a game object and a straight line from point $A$ to $B$.

When running the algorithm we have some settings which is restored after each usage. Initial settings:

- All nodes are initialized with $h = g = \infty$.
- The list of vertices to expand - the open list - is initialized empty.

## 6.3 Final design

The algorithm is designed for terrain with a sparse object population. With fewer objects we get a shorter runtime as the chance of hitting an object blocking the searched path diminishes. When there is fewer objects the pathfinder has to examine and get around it will reduce the runtime significantly as the expand funtion can be called recursively. This is the exact opposite when using the earlier mentioned pathfinders based on a grid layout for the graph. In a grid where a certain amount of space taken by objects the graph will be diminished

[4]A more formal word for the update method is to relax the edges adjacent to the node – in this case we update the nodes found by the expand function

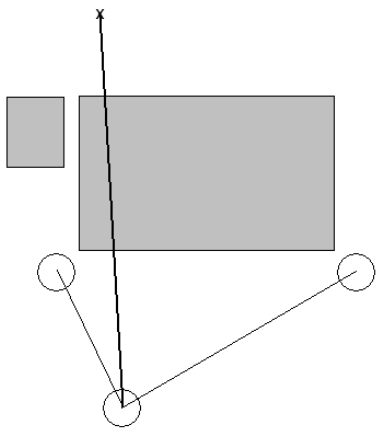


**Figure 6.3:** *The illustration shows the pathfinder tracking around the large object on its way to the target zone. The fastest route however is to ignore the large objects and go straight for the smaller building, around, and then for the goal.*

and making the pathfinder runtime shorter. This make the pathfinder in JWars somewhat specialized as it favourites a certain type of terrain but will still function on densely object populated terrain.

The expand function suffers one fatal error. It *can* fail in finding all the neccessary edges leaving it. An example of this situation is shown in Figure 6.3.

It is clearly that acquiring the nodes on the smaller building would be the fastest route to the target $X$. The path taking the moveable closer to the object however, fits a standard tactical manouvre, where covers means safety from enemy fire. In the real world objects on the battlefield would be used by units to hide their positions or make up defenable position. One other error which can be forced by a programmer is create a single structure from multiple convex hulls. We have already stated that in order to have non-flawed data objects must be convex hulls. If a programmer chose to make create a 'U' formed building consisting of 3 rectangles, the pathfinder would not return a path to the target, merely a path inside the 'U' where it would remain stationary.

The flaw in the expand function could be fixed by adding in a do/while-loop in the update function or a similar fitting place.

```
current = this;
do(
  if(expand(this, current.ancestor))
  {
    this.update(current.ancestor, goal);
  }
  else{ current = current.ancestor; }
)
while{ current != start }
```

Placing this pseudocode in the implementation would make the pathfinder check all nodes leading to node which we just found. It would cut some corners and make the implementation final but have not been included in this final release.

Some pathfinders have been expanded to forecast other units walk patterns, and to take these into their own calculations when searching for a path. 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 never be available to a pathfinder based solely on the terrain objects themselves. In the real world however it *does* make sense not to let *all* allies know where you are *all* the time. This general rule should apply to all RTS games aiming for realism. For solving the issue with units sharing knowledge and optimising paths another type of data would be needed. Implementing a system for units to communicate and plan their movements socially can be implemented. 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 on the small scale. We have experienced some issues concerning two different systems both capable of giving orders to units as they have a tendency to work against each other.

# Chapter 7

# Dynamical game objects

Until now we have described several complex modules, notably the collision detector and pathfinder. This chapter will describe the actual *inhabitants* of this world, how they are organized, which variables they must have and their behaviour.

## 7.1 Unit organization

A central concept of all strategy games is the basic controllable unit, ranging from individual men and vehicles in some games to division-scale (as in the *Civilization* series). The concept of units in JWARS differs fundamentally from the corresponding concepts in other realtime strategy games, borrowing features from turn-based strategy games and real-world military hierarchies. This chapter will provide reasons for and description of the JWARS unit organization and its advantages. The ideas presented below constitute the most important single reason for the existence of JWARS, distinguishing it from all strategy games known by the authors, and this is therefore the most likely feature to make JWARS "famous" if such a thing should happen.

### 7.1.1 Real-world military organization

All modern militaries are remarkably similar in their organizational structure. More or less consistently, the armed forces are divided into several armies which are successively divided into corps, divisions, brigades, battalions, companies, platoons and individual vehicles or squads of infantry. Commanding officers are assigned on each of these levels, and the organizational structure allows large amounts of forces to be controlled as a single entities. The high-level entities are generally referred to as *formations* whereas the lower-level ones (which comprise e.g. purely infantry) are called *units.*

In most cases, each unit comprises three or four units of the next smaller type. For example a battalion might contain four infantry companies plus supporting anti-tank or mortar units. Infantry companies usually consist of three infantry platoons and possible further support. A platoon can consist of three 10-man infantry squads, each man being armed with a rifle except for a light machine gunner and an anti-tank team.

Generally it is practical for the commanding officer at a particular level of organization to directly control units up to *two levels down* in the hierarchy. Thus a divisional commander exerts direct control of a number of brigades, and to a limited degree the battalions. The individual formations and battalions are assumed capable of controlling their own components. It is obviously not practical for a commander at a very high level to control vast amounts of single tanks.

### 7.1.2 Military command in computer games

The category of computer games in which the player controls a large military force with the objective of defeating a similar force in battle can be divided into two primary groups: real-time and turn-based strategy (or tactical) games. In any case the player usually has a *force* which consists of *units*.

Some turn-based games, such as the *Steel Panthers* series, attempt to achieve very high degrees of realism, including realistic weapon specifications, provide a structuring of units into a true military hierarchy, and sometimes these games include scenarios that accurately depict the *orders of battle* (the unit structure and equipment) of the historically involved formations. In *Steel Panthers*, for example, the player has unlimited time to control every single entity no matter the size of the entire army. For very large battles, the player who spends the most time is likely to win. While the units may be organized into platoons and companies, the player still has to control the forces at the single-vehicle or single-squad level, and platoons are thought of as abstract entities and not actually units.

In real-time games the situation is different. First and foremost, the degree of realism is rarely very high, with tanks being able to shoot less than 100 metres and nuclear weapons frequently being a native part of the battlefield. Aside from the ahistorical antics, the controllability of forces becomes *very important* because the player cannot take arbitrarily long time to issue orders. Generally the units are not organized at all, meaning that the player has direct control of every unit. This means that as the game grows in complexity, controlling the units becomes ever more demanding, and the player who is fastest with the mouse frequently wins out due to the better ability to pull wounded units out of harm's way, bring reinforcements forward quickly, and possibly manage resources at the same time.

To facilitate somewhat efficient control, real-time games generally allow the player to *drag a selection box* on the battlefield with the mouse to obtain momentary control of whichever units are inside the box, and every order issued will apply to this selection. Another feature is to organize units into *control groups*, such that the player can use hot keys to select i.e. a group of aeroplanes even though they are not near each other (and therefore difficult to drag a box

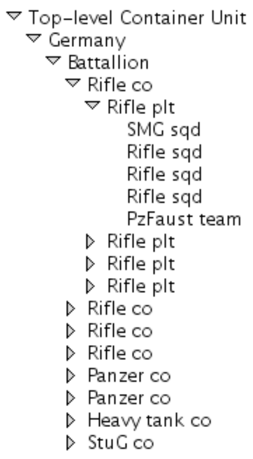


**Figure 7.1:** *Example of a unit tree. Only the nodes with downward pointing arrowheads are expanded. This is part of a screenshot from* JWars.

around). Control groups can be effective, but it can be difficult to manage them particularly if new units are produced continuously, since they have to be manually included in the groups.

### 7.1.3 Tree-based unit representation

Many proponents of turn-based games scoff at the stress and dependence on quick mouse action in real-time games, using nicknames such as *real-time click fests*, while many real-time players find turn-based games boring.

JWars proposes the use of an *explicit* military hierarchy to help control forces of *arbitrary size* in real time quickly and efficiently, reducing the need for quick mouse actions. Since the forces can be almost arbitrarily large, the game world might as well be expanded past that of most games. This will further mitigate the importance of fast mouse action, since the time scales involved in most operations will increase. On the other hand, the reduced reliance on mouse action increases the relative importance of tactical thinking, which will hopefully appeal to both turn-based and real-time players alike.

There is one possible drawback of this model, namely that the structuring of units may not be as the player wants, and that the explicit tree structure lacks the flexibility to use units individually. Nonetheless the structure is identical to that of real military units, which makes it a marketable feature regardless of controllability.

Figure 7.1 shows an example of a military hierarchy in the current version of JWars. This battalion consists of 116 individual entities (vehicles or separate infantry squads), comprising 344 infantrymen and 36 tanks or assault guns.

It has now been established that all controllable entities in JWars should be organized into a military hierarchy. This is the cornerstone of the entire philosophy of JWARS: the player should not *need* to distinguish between controlling single vehicles or larger units such as companies. To help enforce this principle, the concept of a *unit*, which in previous games has always referred to single physical entities (such as tanks) shall in the JWARS context refer to *any* controllable entity.

With this in mind we have defined a base class of controllable entities called `Unit`, which has the two subclasses `Moveable` and `Formation`, where the former represents actual physical entities such as vehicles while the latter represents an abstract concept such as a company or platoon, and can contain any number of sub-units (such as platoons or vehicles, themselves being either subformations or physical entities). Formations and moveables are directly controllable, presenting the same interface to the user.

The game world contains a single unit which serves as the *root* of the hierarchy. Entities can be added to the world, meaning that they are added as sub-units of the root unit. There are presently two teams in JWARS, Germany and the Soviet Union. The teams are examples of formations themselves. Each team contains two battalions, and each battalion is composed of several different infantry and tank companies.

### 7.1.4 Network distinguishability of units

The usage of a root and the unit tree give us convenient references between units and their sub-units. Suppose we want to send a command across the network applying to a particular unit. We must be able to pick out the corresponding units on all clients in the game. A unit is uniquely identified by its position in the unit tree, which makes it unnecessary to devise another datastructure in order to distinguish units over a network.

This relationship has been implemented with a system which we call a *unit tree ID*. Each unit in the game has a unique ID stored in a single integer which enables us to send orders over the network regarding specific subtrees. When an order is given a unit ID accompanies it, and the network ensures that when executed the relevant unit ID is used. The ID tagging is ordered by a single integer split into 6 layers of 5 bits. Each 5-bit layer designates which subformation to choose from the current formation – starting from the root. This means that the limitations on the unit tree is maximum 31[1] sub-formations and a maximum total of 6 layers. It could be argued that a using a *long* would support larger forces yield a more flexible unit tree, but this transformation has not been done yet.

## 7.2 Game data management

This section describes the *data management* strategy used in JWARS. [1, p. 55] defines a *data-driven* system as "...an architectural design characterized by a

[1]If the current layer reads "0" we have reached the wanted formation

separation of data and code". Such an approach is useful for numerous reasons. First of all, trivial matters such as changing the range of a cannon hardly warrant recompilation of the source code. It is preferable that the game content can be changed without even *knowing* the code, such that different people can take care of programming and game content.

This will also make it possible for players to modify the game to provide their own units and weapons. For example, *Warcraft III* is highly reconfigurable and there exist large sub-communities of *Warcraft III* players that play custom modifications of the game[2].

JWars includes a loading routine which reads game data from external files, then converts the data into *categories* which are *factories* for creating various game objects.a

### 7.2.1 Inheritance versus data-based game object classification

JWars contains several different types of units, such as tanks and infantry squads. Further there are different types of tanks, such as *PzKpfw IV* and *T-34*. We note two basic ways of dealing with such variations, inheritance and purely data-based classification.

Common lessons in object oriented programming describe how the abstract class `Animal` could have an abstract subclass `Fish` which could have non-abstract subclasses such as `Anchovy` or `Lamprey`. It would be possible to use a purely inheritance-based hierarchy, meaning that there should be a class called `PzKpfwIV`. But even so there were made variations of this tank. Does this warrant yet *another* level in the inheritance hierarchy?

On the other hand one could use only one kind of unit, then provide a large amount of data to categorize the unit. For example `type=infantry`. The problem is that if flying units are introduced, then every ground unit must somehow state that it cannot fly. This can become very cumbersome.

The natural solution is to use inheritance[3] only in those cases where functionality differs greatly. For example, since infantry squads do not have a turret which can turn around, it makes sense to use a `Tank` class which has one, whereas the other classes need not. The inheritance relationship between different types of units in JWars is seen in Figure 7.2.

### 7.2.2 Category model

Modelling a tank requires a certain amount of data. For example it has a movement speed, turning speed, a cannon, any number (usually two or three) of machine guns, front armour thickness, side armour thickness and the list goes on. It would be inconvenient for the programmer to supply *all* this data every

[2]Notably there are countless variations of "Tower Defense" maps where the players build defensive towers to defeat oncoming computer-controlled hordes, and the widely played "Defense of the Ancients" modification.

[3]Languages which do not support inheritance can use delegation instead

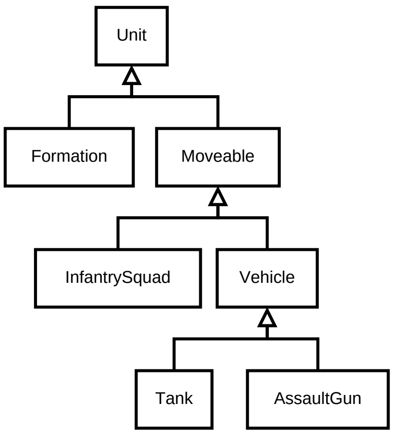


**Figure 7.2:** *Different unit classes by inheritance hierarchy.*

time a tank needs to be created, especially if hundreds of tanks are created, and particularly because most of these tanks are identical anyway.

One solution is to use the *factory* pattern, i.e. a software component which can create any number of units of some type. Suppose every type of unit has its own factory, called a *category*. The category has to contain all the data on which the units of that type rely, but the category does not have to provide any other functionality than that of creating units. By letting units have direct access to their category and its data, they need not store the data explicitly themselves. The categories thus serve as both factories and data repositories for the unit type they represent.

To recapitulate, every unit type, that is, *every configuration of infantry squad and every model of vehicle is represented by a unique category object*: there is a T-34 category for the T-34 tank, a Rifle squad category for the Rifle squad and so on.

Note that when inheritance or delegation is used to distinguish types of units such as infantry and tanks, their respective categories must be able to make this distinction too; it follows that categories should be organized in a similar and *parallel* inheritance hierarchy, see Figure 7.3.

It is not just physical entities (such as tanks) which benefit from using categories. Categories are used to classify all complex in-game components, including tank hulls, tank turrets (it was not uncommon for different turrets to be mounted on the same hull type) and weapons. A tank category, for instance, holds references to its hull, turret and weapon categories. Aside from enabling logical structuring of data, this allows an SU-85 tank destroyer (which historically used the T-34 tank's chassis) to use the hull armour data of a T-34 tank. Also many of the infantry squads in the game use the same rifles, machine guns and grenades.

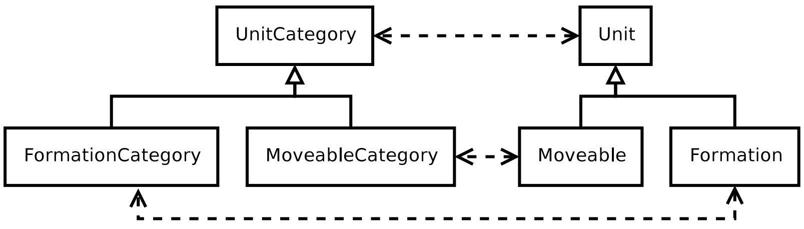


**Figure 7.3:** *Parallel inheritance hierarchy of unit classes in* JWars *and category classes. The fully inked arrows denote inheritance relationship, while the dashed lines denote correspondence between a class of unit and a class of category.*

### 7.2.3 Content loading by categories

JWars provides a *data manager* which serves as a central data repository.

As promised earlier, game content is read from external files. The central data manager can conveniently be used to parse datafiles containing game data, and categories can be created dynamically from data obtained in this way. The datafiles are stored in a custom, human-readable format, see Tables 7.1 and 7.2 which show examples of datafile entries.

When the data manager loads a file, it parses the words in the file (separated by whitespace) in sequence. First it reads the category type identifier ("weapon" or "tank" in the above examples) and uses the type identifier to determine the correct category class (e.g. WeaponCategory or TankCategory). Then the data manager invokes the corresponding category constructor which is responsible for parsing the remaining text from a particular datafile entry. Notice that one of the top entries in each datafile entry is an *identifier*. When the data manager has loaded a category, the category is stored in a dictionary, using the identifier as a key. The category can then be accessed from the data manager by providing that identifier.

Table 7.2, which defines the PzKpfw-IV tank category, holds a list of *weapons*. The weapon names given in the list are the *identifiers* of weapon categories. Thus, as the PzKpfw-IV category loads, it can retrieve the specified weapon categories from the data manager through the weapon identifiers, and get hold of the weapon categories.

When finally a PzKpfw-IV tank is created, the PzKpfw-IV category can use its list of weapon categories to create the corresponding weapons for the tank.

The military hierarchy is similarly created by means of *formation* categories. Formation categories hold references to sub-unit categories (so a company category could hold a list of platoon categories, which could hold a list of infantry squad categories). When the formation category is used to create a formation, it will automatically result in the creation of the sub-units too. For example, creating an infantry company will result in the creation of the four infantry platoons of which the infantry company consists. The creation of each platoon involves the creation of the relevant infantry squads, which again involves the

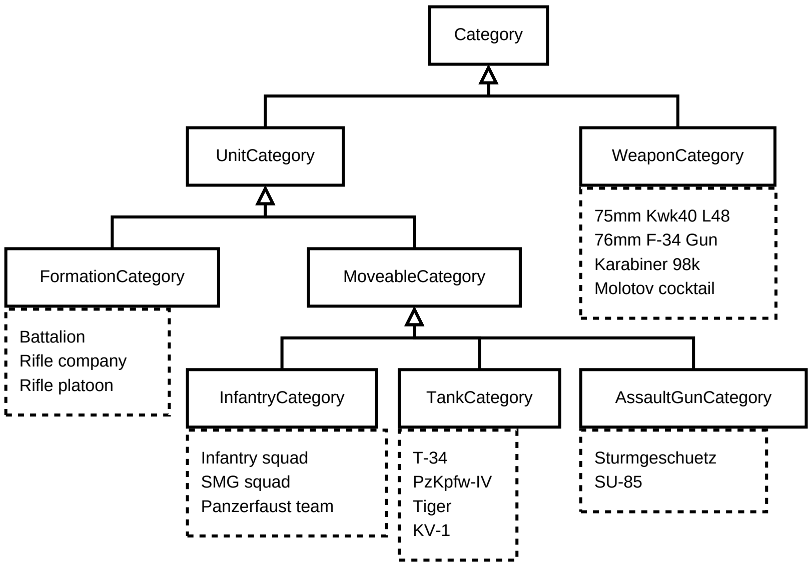


**Figure 7.4:** *Categories. The continuous boxes indicate category classes whereas the dotted boxes list examples of actual category objects of the corresponding class. Arrows indicate inheritance.*

creation of weapons for each squad.

### 7.2.4 Conclusion

We have now developed a system to manage game content by editing text files, i.e. without having to know or touch the code. External data files define weapon types, infantry squads of different sizes using different weapons, and tanks which can be created from a central data repository which is loaded at runtime. The game already includes five kinds of infantry squads, four kinds of tanks and two assault guns. The standard formations, such as platoons and companies, into which the forces are organized are defined in a similar manner.

## 7.3 Unit AI

This section is devoted to the unit AI framework. In this context, AI means relatively simple codes for organized behaviour as opposed to e.g. complex and unpredictable behaviour which may be desired in other games.

### 7.3.1 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

| | |
|---:|:---|
| Type & identifier | `weapon 75mmkwk` |
| Full name | `"75mm Kwk40 L48"` |
| Firing range | `1.2 km` |
| Effective range | `500 m` |
| Reload time | `8.1 s` |
| Firepower data | `ap 120 16` |
| Explosion type | `mediumexplosion` |
| Splash radius | `5 m` |

**Table 7.1:** *The datafile entry defining the weapon category corresponding to a German 75mm Kampfwagenkanone (tank gun). The right column contains the actual lines in the datafile, while the left column is only for description. The firepower data comprises ammo type (armour piercing), armour penetration (in millimetres) and "kill index" (effectiveness against infantry).*

| | |
|---:|:---|
| Type & identifier | `tank pziv` |
| Full name | `"PzKpfw-IV"` |
| Radius | `3.8 m` |
| Speed | `24 km/h` |
| Turn rate | `1.4 /s` |
| *Begin weapon list* | `begin` |
| Main gun | `  75mmkwk` |
| Machine gun | `  mg34` |
| Machine gun | `  mg34` |
| *End weapon list* | `end` |
| Hull type | `pzivhull` |
| Turret type | `pzivturret` |

**Table 7.2:** *Datafile entry defining the German Panzer IV tank. The entries in the weapon list are identifiers of weapons. Notice the identifier of the tank gun from Table 7.1. The other guns and the hull and turret types are also identifiers of categories. These include filenames of images which are used to display the components.*

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 1.2.1). 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 appropriately.

- 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 e.g. 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[4].

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. It is also unlikely that the AI at higher levels of organization can ever outwit a human commander, making sure that human interaction is still required.

### 7.3.2 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 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 therefore reasonable to propose that every unit, whether it is an abstract formation or a physical entity, should possess an AI, and this AI should expose an interface which allows a standardized set of instructions. However the *implementation* of these instructions should be left open, such that the different kinds of units can freely interpret them appropriately.

It further proves useful to have different types of AI specialized in different roles. The code which manages movement not necessarily have much in common with that which manages shooting. Therefore it can be an advantage to hold such functionality separate. Specifically, this will result in a `MobileAI` and an `AttackAI`, each of which provides the corresponding functionality. Since units must provide the functionality of *both*, the logical solution is to assign each unit a `UnitAI` which conforms to the specifications of `MobileAI` as well as `AttackAI`.

This design is obviously well-suited in an environment which allows polymorphism and inheritance, and for this reason the use of Java interfaces are ideal for the core AI classifications.

[4]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 way is faster.

### 7.3.3 AI layering structure

Along with the AI interfaces that specify the AI capabilities, some simple implementations exist which can take care of specific roles. The following example will illustrate the usefulness of this principle.

The `MobileAI` interface specifies an `orderMove` method which is supposed to make the relevant unit move to a specified location. Also similar movement orders can be appended or prepended to a queue of such orders. There is a standard implementation, `MovementQueueAI` which takes care of all this queue management. Suppose now that a pathfinder should be used to break the move order into straight-line segments leading around some obstacles. This functionality can be provided by wrapping the `MovementQueueAI` and providing a `PathFindingAI` with an `orderMove` method which invokes the pathfinder, then enqueues the way points by using the underlying `MovementQueueAI`. The player, however, does not need to know that the AI responsible for pathfinding actually wraps an AI responsible for enqueueing movement orders. The only information which is important is that the AI provides the movement functionality.

In a completely unrelated matter, the `BasicAttacker` which is an implementation of `AttackAI` is responsible for keeping track of a *target* and whether or not to shoot. The implementations which provide movement and targetting functionality can now be reused together. The AI of a physical entity such as a tank (called a `Moveable`) is an implementation of `UnitAI` which wraps a `MobileAI` and a `BasicAttacker`. Thus the behaviour of a tank is dictated by interchangeable AI "building blocks" that can be expanded as required.

This example is of course dependent on the layout which *we* have happened to choose for the AI API, and this might not be what another developer wants. Nonetheless the design shows a flexibility which allows almost arbitrary extensions. In conclusion, units have a particular AI interface which is exposes attacking and movement functionality, and the AI framework relies on *delegation* to various specific implementations to provide this functionality. Interfaces are used for polymorphism.

### 7.3.4 Future AI work

It is no secret that the limited work which has gone into the AI implementations in JWars are not going to revolutionize the real-time strategy genre. However the unique tree-organization allows for much more complex and intelligent behaviour which can be implemented in the future. This section will mention some of the more promising improvements which can be done.

- Aggression *modes*. In some cases it is desirable that units fire at every nearby enemy. But otherwise this might not be a good idea. If a reconnaissance patrol opens fire on the enemy troops they are observing, they will most likely be spotted and killed. If an infantry squad is waiting for an unsuspecting tank to come close enough to throw a grenade down the open hatch, then it is most unwise to open fire at a range of two hundred metres. Thus, a good AI must know when to fire and when not to. When the squad opens fire it is important that the remaining squads of the platoon, or the entire company, open fire as well. It therefore makes sense to make e.g. a company AI responsible for starting such an ambush, though it requires that the AI supports, for example, an ambush state.

- Battlefield-awareness. A common problem in contemporary real-time strategy games is that an airstrike is ordered on an enemy factory somewhere. While under way the planes are attacked by unseen anti-aircraft batteries and shot down. In this case it would be beneficial to call off the attack entirely. But if there is only one anti-aircraft emplacement, and if the attack involves twenty planes, calling off would be silly. Assigning an AI to the entire attack wing would easily provide a means of evaluating and handling such threats.
- Morale-dependent AI. While under fire, people can panic and retreat. This kind of AI could refuse to perform offensive acts if panic sets in. Once again this can be done by replacing the AI implementation temporarily.

# Chapter 8

# Combat dynamics

This chapter deals with the combat model provided with JWars. The combat model encompasses different modules pertaining to weapons and automatic firing routines, armour and damage. Following that, the *vision* model, which is relevant for automatic targetting will be discussed.

## 8.1 Firing and damage

Most real-time strategy games use remarkably similar combat models. Units will fire automatically at enemy units when the enemy units come into range, wait for their weapons to reload and continue firing until they or the enemies die (or until they receive new orders and disengage).

Every time a unit fires, it may or may not hit its target (in many games they will even *always* hit the target), and do damage to the target and possibly the surrounding units based on the weapon used and the type of target.

The canonical way of representing damage and the health of an entity is to use *hit points*. A unit has a certain number of hit points, and every time it gets hit by a weapon, a number of hit points based on the weapon type, target, luck or other factors, gets subtracted. If a unit reaches 0 hit points it dies. The health state of a unit is typically represented graphically by the characteristic green *health bar*, which becomes shorter and changes colour to yellow and red as things go downhill.

This is a very simple basis model which is used in most games. We can mention Warcraft I-III, Starcraft, Dune II, all Command & Conquer games, and the list goes on.

For JWars, however, we have something more ambitious in mind. Reality does not deal in hit points. If a shell hits a tank, one of two things happen: either the shell bounces off the armour doing no or very little actual damage, or else the shell penetrates the armour and will likely cause horrible damage. It does not take 7 hits or 5 hits like in the hit point model, but could take any number of hits. If the tank is sufficiently heavily armoured, no amount of hits from that cannon will destroy it[1].

Such realistic models have been used in the *Steel Panthers* series of turn-based strategy games. Our approach shall borrow some true and tested ideas from this highly realistic series of games.

### 8.1.1 Combat rule set

The combat rule set is the basis for the implementation. This does not mean every implementation has to use this rule set – this is only the default.

- There are two primary types of entities: vehicles and infantry squads.
- Some vehicles are tanks, which have a hull and a turret which can traverse, whereas others are *assault guns* which have a hull and an inflexible superstructure with a cannon. Hull and turret or superstructure each possess an *armour table*, which lists the thickness of steel armour in millimetres and the angle of armour plating. This information is borrowed from *Taschenbuch der Panzer 1945-54*[4] and sometimes *Steel Panthers: World at War*[6].
- Infantry squads have a strength, i.e. a number of men.
- Each entity can have any number of weapons.
- A weapon has a maximum range, an accuracy, a firepower (determining its efficiency against infantry), an armour penetration value (in millimetres of steel, numbers are borrowed from *Steel Panthers: World at War*[6]), an ammunition type and a reload time. A Weapon can fire at a location but is not guaranteed to hit. Weapons can deal *splash* damage, i.e. collateral damage to units near the impact location.
- Whenever an infantry squad is hit or nearly hit by a weapon, people may die depending on luck, impact distance, weapon firepower and possibly other factors.
- Whenever a vehicle is hit directly by a weapon, it might be destroyed based on the weapon's armour penetration ability, the vehicle's armour thickness and the angle of incidence.
- Enemy units will automatically fire at each other if within range.

We intend to expand the ruleset in the future, to support *crewed weapons* (e.g. infantry-operated anti-tank guns or FlaK), *offboard artillery* which can conduct indirect bombardments of any part of the battlefield and *aeroplanes* which are offboard most of the time but can make bombing runs.

[1]Anthony Beevor[3, pp. 90-91] notes a particular occasion on which German panzers fired many shells at an immobilized Soviet KV-1 heavy tank. Finally the Soviet crewmen emerged to surrender, badly shaken, but unhurt.

### 8.1.2 “Weapon vs. armour”, or “armour vs. weapon”?

There is a tricky matter of evaluating different ammunition types versus different armour types which warrants a discussion of the way such checks are handled. This section will discuss real-life weapons systems in order to determine the most sensible way of handling shell impacts.

Suppose a shell hits a tank. We will want to compare the steel penetration of the weapon with the thickness of the armour. If the shell uses kinetic energy as a means of penetrating the armour (e.g. common *armour piercing* ammunition) then its ability to penetrate armour should be reduced with impact speed and thus travelling range. If the shell uses only explosive power (such as *HEAT, high-explosive anti-tank* which is commonly used in infantry anti-tank weapons such as the bazooka, Panzerschreck and Panzerfaust), then its steel penetration is completely independent of impact speed.

The common way of handling such a problem in object oriented languages is to equip each weapon with a different method for calculating damage to steel armour. The problem is that several types of *armour* can also exist, which means the weapon will have to distinguish manually between target types anyway. See below: should the implementation be provided by weapon or armour?

```
armour.calculateDamage(weapon)
//Allows armour class to select implementation
weapon.calculateDamage(armour)
//Allows weapon class to select implementation
```

We have decided that the complexity of *armour* is generally greater than that of weapons, and that the implementation should therefore be left to the *armour* class.

For example, diverse defensive technologies range from no armour (infantry) to steel and *spaced* armour. The previously mentioned HEAT ammunition uses a curiously shaped warhead to achieve a *directed* explosion, forming a jet of molten metal[7] which can travel a certain distance largely unaffected by the type of armour it penetrates. This can be negated by mounting a thin layer of armour on vehicles some distance away from the armour, meaning that the jet will disperse before reaching the inner armour layer. This is called spaced armour. Figure 8.1 shows a Soviet T-34 tank equipped with a mesh to detonate such warheads prematurely. A more modern technology called *explosive reactive armour* or ERA uses explosive charges as part of the tank armour to obstruct the jet, nullifying its penetrative capabilities[8].

Thus we decide that weapons must be characterized by a select few parameters, whereas armour has the benefit of possessing the method which decides what happens on impact, given the weapon parametres. This allows armour systems arbitrary complexity (they can provide any implementation) whereas weapons have to express their efficiency in terms of a pre-determined set of parameters. In order to distinguish different types of weapons (which is still necessary), a few standard types are hardcoded: high-explosive, armour piercing, HEAT and bullets. Bullet type weapons are considered special: unlike the

**Figure 8.1:** *Soviet T-34 tank with wire mesh for protection against the Panzerfaust anti-tank weapon.[9]*

other types, they are considered to fire *volleys* consisting of several shots (such as from a machine gun or a whole squad firing several rifles). Also, if the first weapon declared on an infantry squad has the bullet type, then it is considered issued to *every member of the squad*, meaning it will have its firepower multiplied according to the number of men. The other ammunition types have no explicit meaning, but when calculating damage, the armour can distinguish these types on an if-else basis.

### 8.1.3 Structure of the weapons API

There are four concepts which are introduced in order to properly separate the code.

- `Weapon`. A weapon has a *category* (see Section 7.2) which stores its capabilities, and a *state*, being either loaded or not. The weapon has a *fire* routine which ultimately might result in people getting killed (no humans were harmed during the making of this routine).
- `WeaponModel`. The weapon model serves as an interface between the set of weapons belonging to a unit and the code which attempts to control the unit's more aggressive antics. The weapon model can be used to emulate the weapon set independently of the actual weapons, which allows the weapon code to be substituted without breaking e.g. the unit AI.
- `ArmourModel`. Responsible for handling the (nearby) impact due to the firing of a weapon. Present implementations include two armour models, being infantry- and vehicle-specific, respectively.

- `Damageable`. Responsible for handling any damage caused when the armour model reports that it could not withstand the punishment. Presently this only serves to alert a unit of when it is destroyed, but is supposed to take care of destroyed radios, fire control, suspension, engine etc. if some day those concepts are implemented.

### 8.1.4 Firing routine

The firing routine corresponding to a particular weapon takes the source location and the target location in the main coordinate system as parameters, and validates by checking whether the weapon is loaded and within firing range of the destination. It is desirable, though not presently implemented, that direct-fire weapons (as opposed to indirect-fire weapons which are used for bombardments) should also confirm that they are within line of sight of the target (line of sight is discussed later, in Section 8.2.6).

If firing is possible, the actual impact location is calculated, which can be different from the target location. If the weapon type is "bullet", meaning that it fires a *volley* of projectiles (such as in the case of machine guns), then the hit location is always exactly the targetted location, since this is where the bullets will hit on average. Bullets are then assumed to hit randomly in a "cloud" and not exactly on the central point. Non-bullet weapons have their impact location determined based on luck and the "effective range" of the weapon, but other factors may be included later.

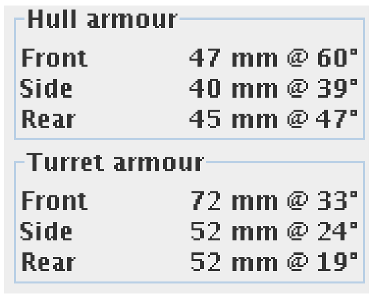
Finally, the set of all entities within the weapon's *splash* range of the impact location is determined by using a utility method provided by the collision detector (Chapter 5) which returns a set of colliders that are within a specified radius from a specified location. All units in this set are considered "hit", though they may not receive any damage.

For each unit which has been hit, the armour model belonging to that unit has its `reportImpact` method invoked, and this method decides what happens to the unit in question.

### 8.1.5 Impact handling by armour

There are presently two types of armour model: infantry and vehicle. As mentioned previously, the armour model determines what happens to a unit when hit. The infantry armour model calculates a number of casualties based on luck, the impact distance and the firepower of the weapon in question. The present implementation serves its purpose but can use adjustments for play balance.

The vehicle armour model is somewhat more complicated. Vehicle armour is specified by *categories* (the concept of which is introduced in Section 7.2). A vehicle is considered divided into three sections: the front, the sides (assuming symmetry) and the rear. For each of these sections there is an armour *thickness* in millimetres, plus the armour *angle* (sloped armour is important in tank warfare and is therefore included in the model). Tanks, also having an armoured turret, have a similar set of numbers for that, see Figure 8.1.5.



| Hull armour | |
|---|---|
| Front | 47 mm @ 60° |
| Side | 40 mm @ 39° |
| Rear | 45 mm @ 47° |

| Turret armour | |
|---|---|
| Front | 72 mm @ 33° |
| Side | 52 mm @ 24° |
| Rear | 52 mm @ 19° |

**Figure 8.2:** *Armour statistics for a T-34 tank. This is part of a screenshot from the game.*

The vehicle armour model first calculates which section has been hit. This is based on the travelling angle of the shell fired, compared to the orientation of the vehicle. For example if the shell is fired from directly in front of the vehicle then it will hit the front armour with certainty, but generally the probability of hitting a particular section of the vehicle is determined (see Figure 8.1.5) by projecting the relevant sections of the vehicle on the normal plane of the impact direction, and the ratio of probabilities is equal to the ratio of projection lengths.

The armour penetration value of the weapon is compared to the armour thickness at the hit location, taking into account the impact angle on the armour plating. For example, if the shell hits at an almost-parallel angle it will have to penetrate many times the length necessary if it hits at a perpendicular angle. More precisely, the effective thickness is equal to the actual thickness divided by the cosine of the impact angle on the armour, further divided by the cosine of the armour slope (i.e. the angle between the armour plating and vertical).

If the armour penetration is still larger than this *effective* armour thickness, the tank is destroyed.

### 8.1.6 Conclusion

We have now explained how weapons fire and how armour models handle the impact from weapons. Weapons are defined by a limited number of variables such as armour penetration, range, reload time and an abstract notion of *type* which distinguishes bullets, high explosives and armour piercing ammunition. Armour models, being more complicated, contain the *code* for handling impacts.

## 8.2 Vision

The last section described how weapons and armour work. But there remains the problem of deciding when to use them. We have mentioned earlier that opposing forces should shoot at each other automatically once the opponents have been discovered.

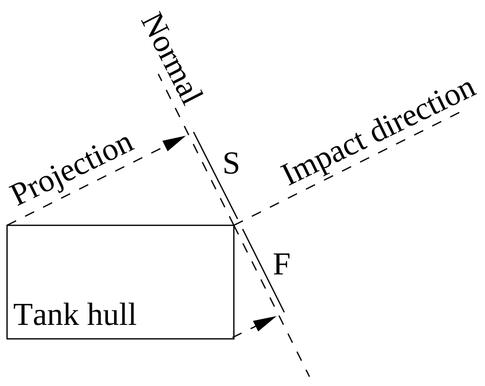


**Figure 8.3:** *Side hit or front hit? The projection* ***S*** *of the tank side on the normal of the impact direction is about equal in length to the projection* ***F*** *of the front, so the probability of hitting the side is about 50 %.*

We have also promised to implement a kind of *fog of war*, a common notion of RTS games meaning that units should not be able to see each other at all time. This section will derive a framework for handling *visibility* of units to opposing units.

### 8.2.1 Vision in games

As mentioned, the concept of not being able to see all enemy units is called *fog of war* in reference to the smoke caused by e.g. artillery bombardments. In some old games such as Dune 2 and Command & Conquer, the entire map is blacked out by the beginning of the battle, and the player has to explore the map in order to locate the enemy. In the two mentioned games, terrain that has been explored once will forever stay visible along with any enemy units in those areas. Newer games generally allow the player *only* to see the immediate areas surrounding friendly units, i.e. as soon as the units move away, the enemy units in that area are once again obscured. In most cases (Warcraft III, Starcraft, Total Annihilation etc.) there is a maximum vision range, which lets a unit observe a circular neighbourhood of their location, except for obstructions of the terrain such as hills or buildings which can block the view. The maximum vision range is usually less than the size of the main battlefield display, for example around 50 metres.

Bearing in mind the realistic approach of JWars we wish a model of vision which can support much larger ranges, namely hundreds or thousands of metres. This is still shorter than realistic spotting ranges, yet *considerably* longer than in contemporary games. Furthermore it should be possible for terrain objects to block line of sight. Finally, we propose that units should be able to hide even though they are well within direct line of sight. It is in reality easy for infantrymen to hide in bushes or high grass (which are not explicit game objects but rather types of continuous terrain, thus not directly blocking sight), and this possibility should be included in any realistic wargame[2]. Thus, there is no guarantee that the player's forces can see an enemy ambush, even though the ambush is technically within line of sight. This "fuzzy" model of vision is common in turn-based strategy games such as the *Steel Panthers* series.

While not yet suggesting a final way to perform this kind of check, the probability of spotting an enemy unit should be larger if the unit is close, moving, large or clad in bright red clothes. It is also possible.

The problem of determining which units are visible to others can be tackled in a number of different ways, which will be discussed in the following sections along with their pros and cons. We shall refer to entities capable of seeing and being seen as *observers*. With two teams in the game, an observer can be in one of two states: it is either *visible* or *not visible* to the opponent. Observers which are not visible to the opponent should obviously not be drawn to that opponent's screen, and his forces should not shoot at an observer unless it is visible.

### 8.2.2 Approach 1: direct observer-observer checking

Suppose that given two observers, there exists a method for checking whether one can see the other (e.g. returning true or false after taking into account a lot of factors). The most obvious way of implementing vision is to continuously check for every observer whether it can see every other observer. This is extremely inefficient because the number of checks increases with the square of the number of units. Actually this is exactly the same problem we encountered when designing the collision detector in Chapter 5. It is obvious that a similar solution can therefore be applied: dividing the map into tiles. If there is some maximal vision range, then a vision check is only needed if two observers are within that range of each other, and if the size of a tile is comparable to the vision range, then it is sufficient to search only the neighbouring tiles for other observers when checking visibility for a particular observer.

Using this method with which we are already familiar, the problem can be solved cleanly and efficiently, provided that the range of vision is relatively short. Unfortunately this may not be the case. In real life the visibility extends many kilometres. Only if the map is much larger than the visibility does this approach yield a significant increase in performance – otherwise we will have to check the entire map or very large parts of the map anyway. It is particularly bad in large battles when many ($n$) observers are within vision range of each other, therefore requiring $\mathcal{O}(n^2)$ checks.

It is clear that the *large* distances involved are our primary problem. However there is an important optimization which can be performed *only* because of the large distances. Over short distances, it is very important that observers respond immediately to spotting an enemy. This is because whoever shoots first

[2]The lack of vision from World War II-era tanks is of particular importance here: infantry units could hide only a few metres away and attack advancing tanks using molotov cocktails, hoping that the volatile fluid would pour into the tank engines.

will likely win. Movement over larger distances takes a long time, and combat is less hectic. Since observation checks generally involve large distances, it can thus be expected that the time scale on which observers will be spotted and disappear is relatively large, i.e. several seconds. Therefore it is not necessary to check which observers can see each other *every* frame. It is sufficient to check once in a while, possibly once per second or even less. Doing this can decrease the amount of time taken by an order of magnitude or more.

### 8.2.3 Approach 2: observer-terrain checking

We have, however, considered an alternative approach which holds some advantages and disadvantages. In most RTS games, units are visible if the terrain on which they stand is visible. Suppose an observer moves. It would be possible to register all the terrain visible to the unit in its new position (this would require a tilemap with a very fine resolution if observers should be able to hide behind buildings, etc.). Any enemy observers within the visible area are then made visible. This means the complexity of the entire spotting functionality for $n$ observers is reduced to $\mathcal{O}(n)$, since work has to be done *only* when observers are moving, and every observer has to do the same amount of work (register surrounding tiles as visible).

As mentioned, a very fine tilemap is required for this approach. With large vision ranges this becomes a problem because of the sheer amount of tiles it is necessary to traverse, namely $\mathcal{O}(r^2)$ where $r$ is the observation radius.

We have selected the first approach because it can be implemented rapidly due to its similarity with the collision detector, because of its relatively high efficiency in most cases (except when very large forces are massed), and because with the suggested optimization it is not likely to be become a bottleneck.

### 8.2.4 The spotting routine

To recapitulate, we have selected an approach to vision checking where all observers should regularly check which other observers they can see. A full vision check, i.e. checking for *all* observers whether which other observers they can see, will consist of a number of separate steps.

For each observer $O$, do the following:

1. Traverse all tiles within vision range of the current observer $O$.
2. For each of those tiles, traverse all observers within it.
3. For each other observer $\tilde{O}$ found, if that observer is within the fixed maximum vision range of $O$, perform an *observer-observer* vision check between the current observer $O$ and the other observer $\tilde{O}$.

The process is illustrated in Figure 8.4.

The exact implementation of the observer-observer vision check is – as we mentioned previously – left open. The particular implementation used in the

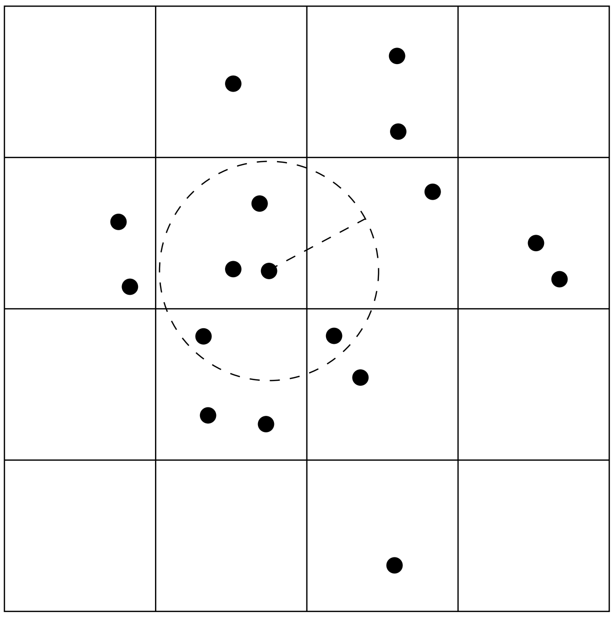


**Figure 8.4:** *The spotting routine. The current observer is surrounded by a dashed circle, the radius of which is equal to the maximum vision range. The algorithm will traverse the four tiles that overlap the circle and expend no time checking the rest of the map. These four tiles contain eight observers aside from the current one, four of which are outside the vision range. The remaining four observers (excluding the current one) that are inside the vision range are subjected to an actual observer-observer check.*

| | |
|---|---|
| Collision detector | Observation environment |
| Collision map | Observation map |
| Collision tile | Observation tile |
| Collider | Observer |
| Collision properties | Observer model |

**Table 8.1:** *Equivalent terms of collision detection and observation handling.*

present version of the game is very simple: the observer-observer collision check simply returns true, i.e. an observer is visible to the enemy if and only if it is within the maximum vision range (which happens to be around 300 metres). The game therefore uses a simple "circular vision" approach.

If an enemy observer leaves the line of sight it should become invisible again. This is a trivial matter if the observer knows when it was last spotted by an enemy – in that case it can regularly check whether the time elapsed since it was last spotted is greater than some pre-defined *relaxation time*, then become invisible as necessary. The relaxation time is presently around 5 seconds, though the exact value is of little importance as long as it is longer than the interval between spotting checks.

### 8.2.5 Final design

Since visibility is handled almost the same way as collision detection, it can hardly be surprising that a very similar design has been employed. Table 8.1 shows an overview of the terms we use in the context of observation, along with the corresponding term from collision detection.

An *observation environment* serves as a centralized manager, hosting the *observation map* which consists of *observation tiles*. The observation environment can register `Observers` which have a location and have the observer-observer `check` routine. `Observer` is technically a Java interface, meaning the implementation of is left open (perhaps infantry squads, having maybe 10 men which can look in different directions, would like an implementation different from that of a tank, where the crew can see only through a small opening unless they open the top hatch). Each registered observer is associated with a particular *observer model*, holding inforamtion which is used "under the hood" (such as the time at which the observer was last spotted).

Note: there are a few differences between the collision detector and vision handling. Whereas the collision detector will register a collider in all the tiles that overlap the collider, the size of the tiles used in this section are very large compared to the actual observers, so there is no advantage in registering an observer in more than one tile at a time. Thus, observers are only registered in the tile at which their center is located.

There is another subtle difference: internally, the collision detector uses an array-based list implementation to represent the colliders registered in each tile.

Removing elements requires that the entries with indices larger than that of the removed collider be copied to occupy lower indices in order to prevent “holes” in the array. For $n$ colliders this takes $\mathcal{O}(n)$ time on average. This is efficient only for very small lists, and indeed collision tiles are not expected to contain many colliders at a time. The important difference here is that the observation tiles are *very* large and can contain hundreds of elements. Thus we have decided to use a linked-list implementation which allows addition and removal in constant ($\mathcal{O}(1)$) time, sacrificing random access which is worthless for our purposes. The *observer model* serves as a reusable link for the linked list, thus eliminating the overhead of creating link objects dynamically.

### 8.2.6 Evaluation and discussion

After having tested the vision system in action with more than 400 observers in one battle, we have not seen any measurable impact on game performance. We conclude that the selected approach is efficient enough for the simple circular vision model used presently and probably (though this has not been tested) somewhat more complicated vision models as well.

It is desirable to improve the observer-observer check to the realism standards proposed earlier in Section 8.2.1. This takes long time to do well, and does not involve any technical problems of comptutationally interesting nature, which is why we have decided not to implement it yet. The implementation should take into account range, the type of unit (infantry can hide more easily than tanks), the speed of the unit (it is easy to spot moving entities), the amount of vegetation in the terrain, and it should incorporate a line-of-sight (LOS) check such that e.g. buildings or trees can block the field of vision. Regarding the LOS check, this can easily be implemented since the pathfinder (Chapter 6) has already implemented the Bresenham line drawing algorithm and used it to traverse the collision map for obstacles. This code can be directly reused to find obstacles to LOS.

# Chapter 9

# Graphics

While graphical beauty is not one of the primary objectives of JWars, the rendering system is designed with some care for performance and practical usability. The system relies on Java2D and the Swing framework, as these shall prove reasonably efficient for our purposes, not to mention the convenience that they are included with the Sun Java Runtime Environment.

There are numerous alternative graphics libraries which could likewise have been used, ranging from the low-level OpenGL wrapper, JOGL[10], to scene-graph implementations such as Java3D[11], Xith3D[12] and the game library LWJGL[13]. In the following we shall discuss a number of rendering strategies with the intent of applying them with AWT/Swing. However, importantly, these terms do not apply only to this framework; they are general principles used in rendering in many different contexts.

## 9.1 Active versus passive rendering

As mentioned in Section 1.1.2, the user interface of real-time strategy games normally consists of a centered main display which displays the battlefield and the animated action. Surrounding this display is typically an overview map and a number of status panels which are not animated, or contain relatively little graphically heavy content.

The main battlefield display will require continuous redrawing due to the dynamical nature of its content, and the rendering operations are expected to be complex and demanding for the computer. Widget toolkits such as AWT/Swing are not designed for this kind of rendering, and it will be necessary to manage the rendering manually: the main display will use *active rendering*, i.e. it will draw directly to the screen when requested, and requests will be issued continuously.

Note that most real-time computer games issue such requests at the maximum possible frequency to ensure the best smoothness of animations. This can be done from a *rendering loop*. We have decided to use a less aggressive approach and render only once every time the logic is updated; this will occur at a 50 Hz rate, which proves sufficiently smooth for a 2D game where most entities move reasonably slowly. However in fast-paced 3D games this is barely considered sufficient by skilled players[1].

On the other hand, since the surrounding panels are not generally animated, these components are ideally represented by Swing widgets using the normal *passive rendering*, where repaints are scheduled as required and taken care of when the computer "feels like it". Since the panels are going to display data which depends on the internal game state and contain buttons which might affect that state, and since AWT/Swing applications run largely from a particular thread, namely the so-called *Event Dispatch Thread*, it will be necessary either to synchronize the interaction between the user interface and the model, or to execute all relevant code in the Event Dispatch Thread. Therefore the entire game logic runs from this thread, but this is of little importance to the remaining parts of the program.

## 9.2 Double buffering

*Double buffering* refers to a technique which can be used to improve the perceived performance of an application. A naïve implementation of a rendering loop would simply clear the rendering surface, then perform the drawing operations and terminate. This will most likely cause the screen to flicker. The explanation is that the drawing operations take so long time that the user notices the screen being temporarily empty. Double buffering uses two drawing surfaces: a *on-screen* buffer which is displayed, and an *off-screen* buffer which resides somewhere in the computer (or hopefully the graphics adapter) memory. A graphical update could consist of clearing the off-screen buffer and performing all the rendering operations onto it. Then the off-screen buffer is drawn (or *blitted*, a particular technique used for rendering images) onto the on-screen buffer, making the changes visible in one sweep. The blitting can even be synchronized with the refresh rate of the screen, though we shall not go into detail with this.

There are other techniques associated with double buffering, for example *page flipping* which interchanges the off-screen and on-screen buffers simply by switching a pointer. There are approaches that use even more buffers, although this is hardly of interest here.

Swing applications are automatically double buffered. Only the main display, which is actively rendered (and which therefore does not use the Swing repainting mechanisms) cannot automatically be double buffered. Implementing proper double buffering would require the allocation of the aforementioned buffers, preferably in video memory. Fortunately this is not necessary in our *particular* case because AWT happens to provide a `Canvas` class which can have its own `BufferStrategy`[2]. Double buffering is hence of little practical concern,

[1]It is commonly known that televisions use much lower framerates. Smoothness is in this case achieved because the frames are blurred and perhaps interlaced.

[2]A Swing-competent reader might notice that the `JFrame` can likewise use such a `BufferStrategy`. But doing so would affect the passively rendered panels in the GUI as

though it remains important to any rendering system.

## 9.3 Battlefield rendering and layers

As it has previously been explained, the primary display shows some subset of the battlefield, the content of the *viewport*, in high detail. There are several types of graphics which are to be displayed here, and it will prove advantageous to organize them in *layers*.

1. First, there is the ground terrain. As described in Section 4.3, the terrain is represented by a tile map of terrain tiles, called the terrain map, and each such tile is capable of drawing itself to the screen (provided an AWT graphics context). Not all of the tiles need to be drawn – see Section 9.4.
2. The next step is to draw all the ground units, e.g. tanks and infantry. Since it is cumbersome to traverse *all* existing entities and determine whether they are inside the view, the collision detector comes in handy: converting the viewport bounds to collision grid coordinates allows the traversal of only those collision tiles that overlap the viewport, and thus cleanly provides all the entities to be rendered. Each entity, being a so-called *sprite*[3], is responsible for painting itself given its screen coordinates.
3. Having painted the ground and the entities on the ground, the next level is vegetation (which is presumed to be taller than those entities). Each terrain tile is capable of drawing its vegetation to the screen, and this will overlap any units present[4].
4. When cannons are firing, there should be explosion animations to designate the locations of impact. These should be visible to the player (even if physically situated below trees) since they provide valuable information. There might be rockets or aeroplanes flying through the air. All these things (although neither rockets or planes exist in JWars yet) can all be rendered together. While airborne projectiles should theoretically be rendered ordered by their altitude, this would be troublesome, and even when aeroplanes are implemented in JWars, there will hardly be sufficiently many of them so close together as to warrant such an ordering.
5. Finally it might be desirable to display information such as text in the main display. When a unit is selected, a green line indicates its direction of travel, whereas a red line indicates its target. These effects which are

well. Only the `Canvas` offers the desired control over the rendering process.

[3]Sprites are single, flat graphical components such as images or animations, several of which can be drawn together in a context. In two dimensions it is difficult to create something which is *not* a sprite. In three dimensional games, however, sprites can be used for e.g. smoke which has no need for a 3D structure.

[4]When an entity stops moving it will be drawn on top of the trees. This makes sure that the entity cannot go "missing" in the woods, which would be a serious moment of irritation for the player. Interestingly, this feature was originally a glitch in the rendering routine.

not physical entities serve to enhance the ability of the player to control the forces. Their purpose is to convey information to the player without otherwise obstructing the battlefield view. We shall refer to this kind of effects as the *Head-up display* or *HUD*. This type of display is commonly used in military aeroplanes and computer games.

Some of these layers will mostly have stationary content, such as the ground and trees, the display of which should be updated only when viewport is relocated. Others will have dynamic content, such as explosions and moving entities. The following section will provide a solution to rendering these layers *efficiently* taking into account their differences.

## 9.4 Optimization of the rendering routine

Obviously, a battlefield display in which no movement occurs needs not expend any resources rendering. However if a car is driving across the screen, the area immediately around the car will need to be updated as it moves. The *terrain dirtification system* is designed to take care of this, ensuring that minimal time is used to needlessly render terrain.

Whenever an entity moves, the collision detector is responsible for traversing the area and checking whether the entity collides with others. Suppose every *terrain tile* in the terrain map can be in one of two states, either dirty or not. The collision detector can then traverse the *terrain* tiles overlapped by the *sprite* belonging to that entity, and set the state of these terrain tiles to *dirty*, signifying that the tiles need to be redrawn. This will allow the painting routine to filter out those tiles that are dirty and paint them, ignoring the rest. After having been painted, the tiles are no longer considered dirty.

There is one problem with this approach: while it will accommodate the first three layers, the dynamical content such as the HUD cannot be rendered in this way, because the collision detector does not (and should not) know about this. This will result in the terrain not being repainted while the HUD changes, thus leaving graphical artifacts on the display.

Our solution is to render the first three layers onto a secondary off-screen buffer (which needs only relatively little repainting work). The secondary off-screen buffer is – every frame – then rendered onto the primary off-screen buffer which we introduced in Section 9.2. Finally the remaining layers, which generally need complete repainting for every update, are rendered onto the primary off-screen buffer, the content of which is finally blitted to the screen.

While the introduction of this extra step takes *some* time, it still yields much better performance. Drawing an image (such as the secondary off-screen buffer being drawn onto the primary one) is a very fast process, whereas the remaining in-game graphics, involving rotations and possibly transparency, are much more time consuming. Modern computers are capable of rendering images (without e.g. rotation) hundreds, possibly thousands of times per second depending on resolution, and normally this process takes place in the graphics adapter and therefore requires no actual CPU activity.

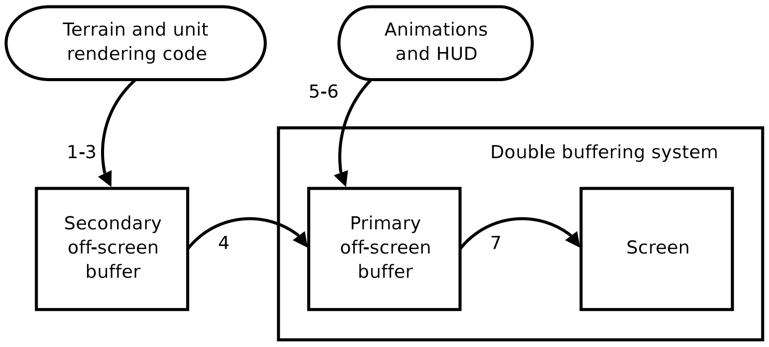

Terrain and unit rendering code
Animations and HUD
5-6
1-3
Double buffering system
Secondary off-screen buffer
4
Primary off-screen buffer
7
Screen

**Figure 9.1:** *The rendering routine. Different steps are indicated by numbers. Steps 4 and 7 are very fast on modern computers and are not likely to have significant impact on performance.*

Finally, let us summarize the complete rendering routine.

1. Render any dirty terrain within the viewport to the secondary off-screen buffer.
2. Render any dirty entities within the viewport to the secondary off-screen buffer.
3. Render the vegetation of any dirty terrain within the viewport to the secondary off-screen buffer.
4. Render the secondary off-screen buffer to the primary off-screen buffer.
5. Render any animated effects onto the primary off-screen buffer.
6. Render the HUD onto the primary off-screen buffer.
7. Render the off-screen buffer onto the screen.

Figure 9.1 shows a visualization of these steps.

## 9.5 Conclusion

In this chapter we have derived a double buffered active rendering routine for two-dimensional top-down view game graphics. The routine saves time by using a third buffer to keep track of the areas on the screen in which no movement occurs.

# Chapter 10

# Game improvements

## 10.1 Future work

Here we will list areas designated for future improvements by the developers and known bugs in the version of JWars following this report. Here is a list of features and areas planned by the developers

1. Terrain heights in the world. Hills should be implemented as fast as possible for tactical gameplay

2. Line-of-sight should form direct controls of forestation and other terrain obstacles for tactical gameplay

3. Offboard artillery

4. Fix current onboard artillery to something usefull

5. Air bombardement and AA guns should be implented

6. Different formation patterns and GUI to support them – instead of the arrow formation

7. Moving formations would make all sub-formations move with the same speed

8. Enable replays – save all registrered orders in a list

9. Enable more terrain as sand, water and jungle

10. Night/Daylight combat mode

11. Better selection of targets by the AI

## 10.2 Known issues

This is a list of known bugs in the current implementation.

1. When ordering formations to move it is possible to cause an `IndexOutOfBoundsException` by moving large formations near the maps edges – some units will get orders outside the map. This can be corrected by translating off-board locations to sensible locations in the relevant code, or by simply removing units that leave the map. We consider this a bug in the code which uses the collision detector, and not the collision detector itself.

2. When moving a unit away from a terrain object while within the same collision tile the pathfinder thinks the terrain object lies in the path of the unit and will return invalid move orders (no exceptions).

3. Units do not reset their targets properly when the targets move out of vision range. The unit keeps firing at the now invisible target.

4. The pathfinder presently cannot use mobile entities as obstacles. A rudimentary system which handles collisions between units on an AI level will give move orders along with the pathfinder – conflict. This is solved by expanding the pathfinder to work on moving entities so only one authority is needed for moving units.

# Chapter 11

# Conclusion

The purpose of this project was to create a real-time strategy game which borrowed elements from turn-based strategy games and implemented a realistic military hierarchy allowing better control along with a level of AI group coordination which is potentially unmatched among games of the genre. The hierarchical structure has not been used before. The AI framework makes heavy use of polymorphism and delegation to provide the possibility of dynamically changing AI behaviour that can be made to adapt to changing tactical circumstances. The current implementations of AI are only rudimentary and do not actually interact, but the framework ensures that relevant AI code is invoked when appropriate.

The most important API packages that are necessary for such a game are fully functioning except for minor bugs. The modules are designed specifically for large game worlds and have proven efficient, even without extensive optimizations, in handling hundreds of units engaged in battle at the same time during network play. Collision detection and vision management use tiles to localize entities and reduce algorithm complexity while retaining flexibility, allowing units of arbitrary size. Special care is taken to allow programmers to implement custom code for handling collisions and custom visibility checks while the framework can invoke the provided code as required.

The pathfinder employs an approach specialized for large maps which generates a search tree dynamically, being particularly efficient in open areas. This can reduce memory consumption and increase efficiency for large game worlds compared to conventional approaches using inflexible high-resolution grids.

The combat system is highly generic, and while the present implementations still need game balancing, the system uses particularly realistic armour and weapon representations such as in many advanced turn-based wargames, thus attempting to create a hybrid genre.

Most game data is loaded from external files using simple script-like syntax which can be modified by people without knowing the source code. This model is ideal for normal game development where coders and game designers work separately.

A two-dimensional rendering system has been developed to efficiently manage tile-based maps by avoiding unnecessary repaints through the use of separate off-screen buffers.

The networking code is based on a client/server architecture and requires very small bandwidth. The game has been tested on Sun Solaris 10, Microsoft Windows XP and Ubuntu Linux 6.06.

The goals defined for the project have thus been reached, except for a single feature we have not had time to implement. JWars regretfully lacks the notion of terrain height.

# 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)

[3] Antony Beevor, *Stalingrad* (Penguin Books, 1999. ISBN 0-14-024985-0)

[4] Senger u. Etterlin, *Taschenbuch der Panzer 1943-1954* (J.F. Lehmanns Verlag München, 1954.)

[5] Wikipedia entry on professional Starcraft competition.
`http://en.wikipedia.org/wiki/StarCraft_professional_competition`

[6] Steel Panthers: World at War by Matrix Games.
`http://www.matrixgames.com/`

[7] Wikipedia entry on shaped charge ammunition.
`http://en.wikipedia.org/wiki/HEAT`

[8] Wikipedia entry on explosive reactive armour.
`http://en.wikipedia.org/wiki/Explosive_reactive_armour`

[9] Wikipedia entry on the T-34 tank.
`http://en.wikipedia.org/wiki/T-34`

[10] JOGL – Java OpenGL bindings.
`https://jogl.dev.java.net/`

[11] Java3D, scenegraph based 3D API.
`https://java3d.dev.java.net/`

[12] Xith3D, scenegraph based 3D API.
`http://xith.org/`

[13] LWJGL, Light-Weight Java Game Library.
`http://lwjgl.org/`

# Appendix A

# Game manual

Before running the JWARS program there are some requirements which must be met by the computer. We haven't tested the application on slower computer systems, but we know that the following requirements are sufficient.

- Java Runtime Environment 1.5.0
- 1200 MHz
- At least 50 MB free RAM (incl. virtual machine)
- One network port (7777 by default) must be available to run the program in multiplayer

## A.1 Running the program

To run the program you will ned the jwars.jar file which can be downloaded for free at

```
http://www.student.dtu.dk/~s021864
```

This is the homepage of Ask Hjorth Larsen, one of the developers. The homepage will have the newest stable version ready for download at all times.

Having a Java Runtime Environment installed, the game can be run by doubleclicking on the .jar file in Microsoft Windows or by using a similar function in other operating systems. Starting the program this way will run JWARS with the default settings. By using a *command prompt* it is possible to run the program using parameters which changes screen size, looks and other options using the command

```
java -jar jwars.jar <parameters>
```

in the library containing the jwars.jar file. The parameter string consists of a single dash followed by a number of letters. Here is the full list of available parameters of the current version:

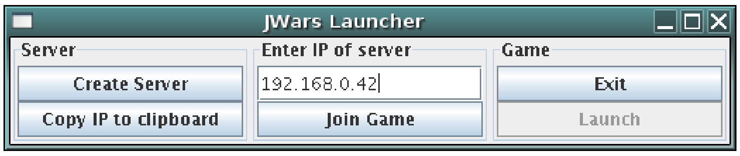


**Figure A.1:** *The* JWARS *launcher.*

- o : Enable OpenGL pipeline. This greatly improves performance, but does not work on all graphics adapters.
- f : Run in full-screen mode.
- h : Print this help and exit
- m : Use Motif look-and-feel
- n : Use native look-and-feel
- d : Draw debug collision grid
- a : Bad ATI driver mode (disable window decorations)
- v : Print version information and exit

For example, the command

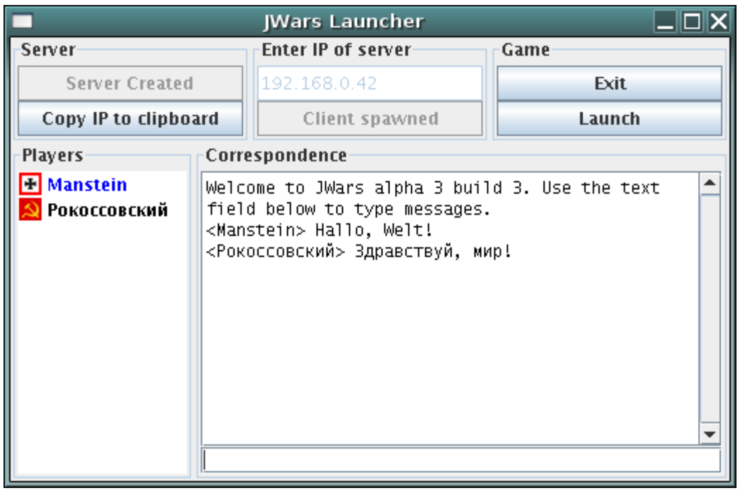
```
java -jar jwars.jar -ofm
```

will run JWARS using the OpenGL pipeline in full-screen mode with the Motif look-and-feel.

On starting the program the JWARS *launcher* will appear, see Figure A.1. The launcher is a tool for setting up a game in multiplayer by either creating a server or joining a server specifying an IP address. The default entry in the IP field is *localhost* which will attempt to join a server created on the local machine (this is useful for testing network support when only having access to one computer). In order to join games over the internet simply enter the IP address of the creator in the IP field and push join. If succesfull you will join the game at the wanted ip address if not an error message will be displayed. It is not possible to connect to a server who runs another version of JWARS.

If a new game is wanted simply press 'Create Server'. Creating a server will expand the JWARS Launcher to a lobby where all currently connected players will be displayed. The lobby can be used for exchanging messages in order to set up teams or to simple correspondance, see Figure A.2.

In the lobby all connected players will be listed in the left side. The game supports any number of players but only two teams. A player can select his team by left-clicking on his name. By doing this the team will change to the opposite flag. When joining the default name will be Manstein and team will be



```
Welcome to JWars alpha 3 build 3. Use the text
field below to type messages.
<Manstein> Hallo, Welt!
<Рокоссовский> Здравствуй, мир!
```

**Figure A.2:** *The game lobby. Manstein and Rokossovsky are chatting before a friendly game of* JWARS.

Germanny. The name can be changed by right clicking on a players own name - in this case a text box will pop up and ask for the new name.

The game creator can start the game at anytime by pushing the *Launch* button. When launching the launcher itself will close and the game GUI will be opened in a window.

## A.2 In-game control

Now we will describe the JWARS GUI and how to use it. First we will focus on the different panels and how to use them for getting information and then how to manage the units under your control. Most of the players interaction will done by the mouse. The keyboard offers certain actions but JWARS can be played without using the keyboard.

### A.2.1 Using the panels

When the game is running the GUI will offer the player all necessary tools for gathering information and controlling the units, see Figure A.3. In order to get an overview of the current positions and forces, look at the minimap placed in the bottom left corner of the screen. To manoeuver around on the minimap simply left click somewhere on it and the focus will move to that location. Using the mouse on the minimap to move the main screen in the game is an efficient way to cycle around the battlefield.

Holding down an arrow key will make the main view scroll in the arrow key's direction. The minimap will show all known unit location in color code (red for russian army and blue for german army). Forests are dark green. The enemy is

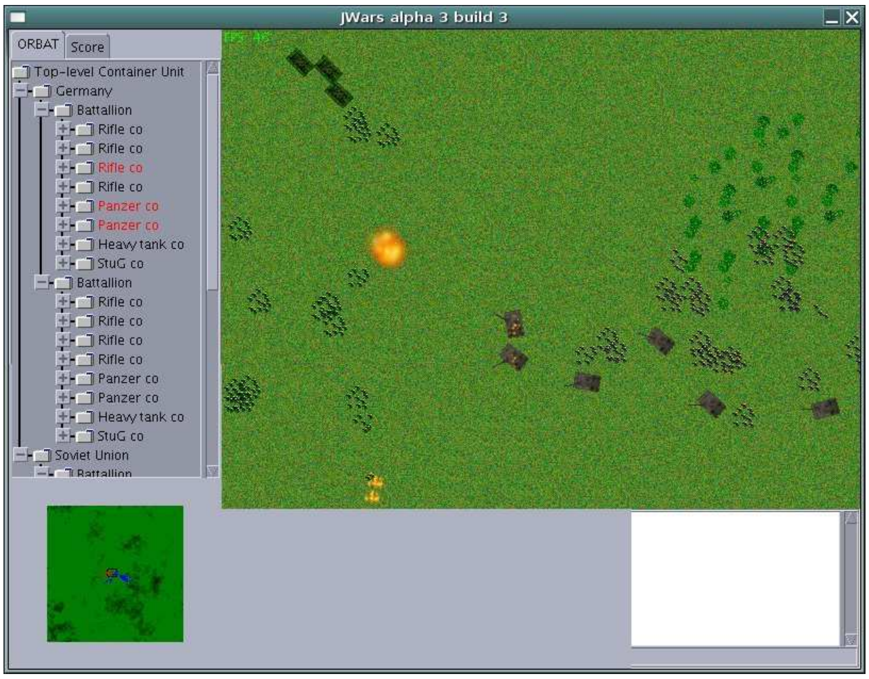


**Figure A.3:** *The battle is raging between the Russian and German forces.*

most likely not visible from your start location, so there should be only either red or blue forces visible presently.

The lower right corner contains a command line. Messages typed here will be sent to all players in the game, unless they start with a slash character, in which case they will be interpreted as commands. Here is a list of the usable commands.

- /commands : Writes a list of different commands for the command line.
- /time : Prints the current time.
- /countunits : Prints the total number of selected units or, if no units are selected, prints the total number of units in the current game.
- /lateness : Prints the number of milliseconds which the game is currently behind schedule (under normal circumstances this should be no greater than 20).
- /exit or /quit : Quits JWars
- /clear : Clears all text from the console

The upper left panel contains the ORBAT, or *order of battle*, which is a list of the available units on each team. Click on a particular unit in the tree to select it. The tree view will automatically expand nodes to provide information about the selection. Units that are killed will have their names written in red.

Clicking on the 'Score' tab will show the current force strengths and casualties in terms of vehicles and infantry.

The bottom middle panel contains information about the currently selected unit, or is empty if no unit is selected.

If the selected unit is a formation, this panel will list its sub-units. If the selected unit is a single entity it will list the weapons of that entity. Clicking on a weapon in the list will write the weapon data to the console. If the selected unit is an infantry squad, this panel will also show the number of men. If it is a tank it will also show the armour thickness and angles on different parts of the tank.

### A.2.2 Marking and moving units

When a player wishes to move units the relevant unit must be selected first. There are several different ways of selecting units where each can fulfill a certain need for a situation.

The simplest way to select an unit is by left-clicking with the mouse on it in the main display. In doing this the unit under the cursor will be selected as the only unit. Double clicking on the unit will select that unit's *superformation*, i.e. its platoon. Triple clicking will select its company, and quadruple-clicking selects the entire battalion. It is not possible to select formations larger than a battalion.

There are several ways to select the super-formation of a unit besides clicking multiple times on a unit. While having a unit selected, rolling the mouse scroll wheel upwards will successively select larger super-formations.

Pushing the backspace button has the same effect as mouse wheel up. Each additional order to mark the superior formation will move the selection *one* step up the chain of command. This is an unique selection command for JWars provided by the special unit tree. This brings to the next speciel entry in JWars.

On the left side of the screen is the unit tree. The panel is a tree view of the formation structure which can be expanded and minimized to provide detail or overview. Clicking an entry in the unit tree will also select that formation or unit and center the view on it.

When having selected the wanted formation or unit simply right click on either the minimap or in the main panel. This will make all the selected units move to the selected spot in formation, i.e. not all standing on the same spot. As units move across the map they may eventually spot enemy units and will open fire on the enemy if possible. This might result in units dying, being removed from the game, possibly in a big explosion. Units that are destroyed can no longer be seen or controlled.

The game does not presently end even if a force is decimated.

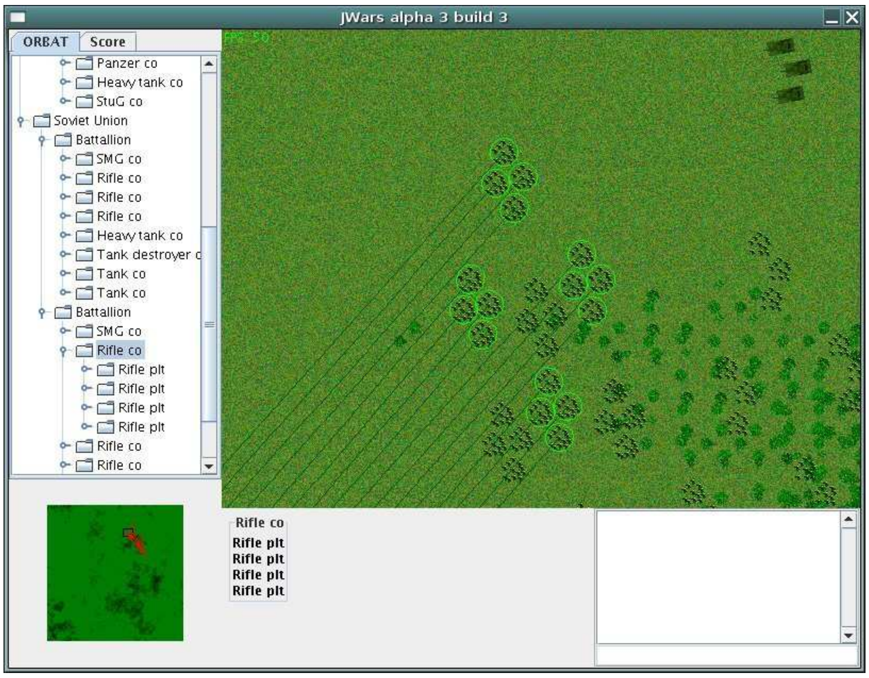


**Figure A.4:** *A green circle around units is a notification about the players current selection of troops. When troops are selected their destination coordinates become green lines on the ground to illustrate their current heading.*

You can order the selected units to fire manually at a location by holding shift and pressing the left mouse button on the desired location.

If any questions arise when playing, feel free to email either of the developers on emails `asklarsen@gmail.com` or `michael_francker@hotmail.com`. Questions will either be answered directly or by reference to this paper which page and line number.

# Appendix B

# Development plan

During the time developing this project we have had numerous versions of the game. We have used these earlier versions for creating a development plan which demonstrates the different stages the game has been through. In doing this we have created a development timeline. Before the project started we had a list of features we wished to implement in JWars and in the end of this section we will evaluate this list to the final product.

In this section we will take the most important aspects of each stage of the project, and supply them with an identifier, marking the element's status at that stage of development: "−" denotes the element as unfinished and requires more work, "+" means that the element is in an acceptable state for delivering but further development is open and "✓" implies that this part of the program is as finished as it will ever be.

The focus in the first stage of development was having a world in which to play and test JWars. One of the important features was being able to play the game online so a server/client model had to be running. This is because we expected synchronization to be difficult and wanted it to work from the beginning. Having an early version of the network it would also be possible to adjust the model to later needs. Another important aspect of the game to have in a useful state early on is the GUI. No final GUI model or look was chosen at that time but we still needed it to test the network and world code.

− Simple working GUI ready for extension

− A unit representation

− Element of control – mouse listener

− Networking code, Server/Client relationship inc. Timer

+ World buildup – Coordinate systems and tile maps

+ Command line/Chat panel used for debugging

When the elements above where working in union we had a base to build on and could now extend the individual parts of the implementation.

The second stage of development centered around building frameworks. During this time some needs became obvious for further development and key areas were created. Especially the collision detector took time during this stage.

– Moveable/Formation/Unit framework

– Basic AI framework

– Terrain dirtyfication framework

✓ Event handling framework

✓ Collision detector

As the game began to take shape updates on certain areas was needed. While still extending the game engine the game content began reaching a satisfactory level.

+ Extended AI framework – Interface set

+ AI implementation of low level collision handling

✓ Networking improvements (client/server event handlers)

✓ Better support for multiple unit types – data managament

✓ Support terrain implementation – forestation and objects

✓ Terrain generator – Height generator tool

✓ Rendering mechanism improvements - secondary buffer

At this stage the game was playable and had fulfilled the minimum requirements stated before the project was started. With multiple types units and an actual terrain to play on, the game looked nice and simple (with room for improvements). At this time we were nearing the date of delivery and the remaining important game features would have to be implemented. The features implemented during the last and fourth stage are important for any RTS game, and we focused on finishing these particular features instead of expending time on e.g. game content which is less critical.

+ Combat dynamics

+ Final GUI layout

✓ Fancy graphics (explosions)

✓ Pathfinder

During the fourth stage we managed to include most of the wanted features for JWars. There is one feature we hoped for which did not get to implement, and that was the concept of terrain height.