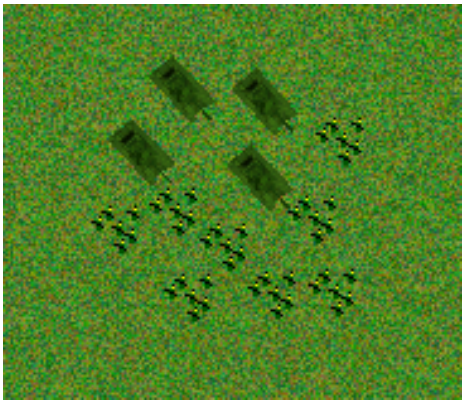
JWars - A Generic Strategy Game in Java

Midterm project — Informatics and Mathematical Modelling

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

Supervisor:
Paul Fischer

August 1, 2006



DTU - Technical University of Denmark
Lyngby

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

Abstract

Contents

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 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

Before reading on in this document a formal introduction to the *real-time-strategy* (from now on referred to as RTS) genre can be neccesary. This section should be seen as history of the genre as well as a opportunity to understand the generel game structure as well as the more advanced concepts in the genre. First we will define the genre and then a quick walkthrough around the 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

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

1. Warplanning is essential – strategy
2. The player has no 'Next turn' button – real-time

Other essential guidelines:

1. Resource gathering
2. Building/unit locations are essential
3. The manufactoring of specific units
4. The player has direct control of his units/buildings

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 for which the developers was inspired by Sid Meiers Sim City. It should be noted that while Sim City differs from the standard RTS game, it is also recognized as a RTS game where the opponent is the game environment itself and not an AI or another human player. As such many diversities has risen in the RTS genre as game developers become more inventive. Today RTS games are in general build on a player vs player environment with single player campaigns for the different races/factions.

Most strategy games requires the player to understand basic military concepts and most often a paper-rock-scissor approach on unit combat. A unit can defeat some opponent units, while it in turn will be defeated by a suitable opponent unit. Often this is combined with a development in the players armory for the cost of resources and time. Resources is 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 have been the basic approach to strategy games, gather resources, build up military forces, gather more resources or focusing on cutting of the opponent resource income. In this cocktail of choices for the player comes the tactical maneuvres 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 maneuvres, entrenchment and so on. As the computer game industri grows so does the amount of time and money spent on developing new features in strategy games. Many of the more succesfull games has found a firm middleground in supporting alot of features but not making the game dependent on these. This will allow more simple users capable of enjoying the game in a more relaxed playstyle while the hardcore gamers can dive in to micromanagement of troops, exploitation of game engines etc.

The average RTS game normally uses the single player campaigns as a linear story introducing more and more different 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 completes more 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 explore game features in turn, thus not making the game seem to complicated. In the JWars project however we will not be including single player missions as it would be beyond the project scope setting up scenarios.

In the last couple of years RTS games has 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 the Blizzards release of Warcraft III: The Frozen Throne did the standard graphic engines change to 3D. Graphichs influenced some games popularity though most is based on gameplay and the univers in which the game takes place. Almost all newer titles uses 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.

1.1.2 RTS combat and control

RTS games focuses on large scale combat. All actions made by a player is primarily made with the thought of hightening his combat efforts. With this in mind an example of unit balancing and a brief explanation of a GUI will open some doors for the inexperienced players. We have chosen these specific areas due the normal lack of understanding in them. 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. Some players create a ratio between units in heads-up combat like 2:1 or 7:2 as this would represent unit data in its rawest form when comparing. In this instance we generalize the concept for better understanding. If we create an example with 3 different units being meassured against each other for example: a plane, a tank and a anti-aircraft gun (AA gun). Logic would create simple rules from this setup:

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

We could attach a ratio on each instance if we wanted to use a measurement. 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. Unit balancing is one of the greatest for developers and is often an ongoing proces after the game has been released. Games today which base their playerbase on an online environment has the ability to release updates when needed. More often than not the developers will release a game which is unbalanced and only the testing done when selling 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 Blizzards Starcraft including the expansion pack - Starcraft: Brood War. This game has been chosen because it is seen as the best strategy to date by most fans of the genre and because both writers of this paper is proficient in this game.

An easy way to spot a RTS game is often by the user interface provided. Several designs with unique abilities and setups have come up but most contains the three most common features - a minimap, an infopanel and a focuspanel,

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

commonly named status-panels as a group. These are all tools for the player to enhance his control and the ability to gather information developed for easy access and usage.

The GUI is split into different subsections each providing the player with information and options.

As seen on this screenshot from Starcraft: Brood War the minimap is located on the bottomleft. The minimap is the primary source for the player to switch his focus on the battlefield as well as obtaining a quick overview. The minimap usually shows the players own forces in green and opponents forces in red. In this way an enemy force massing forces or approaching your territory will result in red markers on the minimap. This is an effective way to aid the player in sorting the information from on screen. The minimap will never be the players main source for information as the information it provides is always sparse and can even be misleading.

Covering most of the screen is the focuspanel[1]. The focuspanel will normally be where the players attention will be centered. The panel is a zoom in from the minimap, but where the minimap only gives the most basic information this panel gives the player a detailed overview of the area. He can point out specific unit types, quantity etc. The panel is simply showing the area of the players current focus but is often used for evaluating tactical situations. Often the size of the focuspanel is balanced out with the standard size of a battle in the given game for easying the players control.

The infopanel is a tool used by the player to gain optimal information about any given object on the battlefield. When a player has his docus 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

[1]This is the most commonly used display for information and can also be called primary display or main display

it will often display a single units statistics and current status.

The user interface in starcraft is a standard example for the genre. The simpel three-step-information-interface handles most situations very well and this setup is used by most RTS games today. When the player can select where, which and a degree of information he want, only poor handling of the system can make it difficult to use. Most new players to a RTS game has a tendency to use the focuspanel as the only source for information while multitasking between all three is a must for players who wish to win.

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 of contemporary games, then we shall see how these might be remedied.

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* (see Section ??) 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. Table **??** shows the organization of something something **FIXME**[2].

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

1.3 Report overview

[2]fixme: consult Antony Beevor's book and insert stuff

Chapter 2

Features of JWars^{TM}

2.1 Game dynamics

game design regarding hierarchy

2.2 Technical features

This section lists briefly the

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

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

Chapter 3

Overview

For reasons of extensibility, JWars consists of several modules which can be used separately or with a minimum of cross-package dependencies. The following chapters will describe each of these modules in turn, but in order to achieve a overview of the structuring of these modules in an actual game, we shall here list the main modules and then describe their high-level interaction.

3.1 Development plan

3.2 Modular overview

Describe basic concepts such as units

Chapter 4

Architecture

In this chapter the architecture**FIXME**[1] 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.

4.1 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 than 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 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[2].

[1] fixme: is this actually the architecture?

[2] 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

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 sted also involves registering the *root unit*, which is the ancestor in the tree hierarchy of all units (see Section **??**) which will later be added to the world.

The following step reads all unit, weapon and formation data from external files (though can 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 will be represented in *category* objects, which hold data pertaining to specific types of unit. For example, the information of a Panzerkampfwagen IV is read once, and then scores of panzers can be spawned using the category as common data.

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.

At this point the entire game setup has been loaded, but the game has not yet started. Rather 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.

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.

4.2 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 move (using the collision detector), turn around, take aim, fire and so on. Specifically, the *update* method of each unit is invoked recursively down the unit tree. This will also perform various other tasks, such as polling for network input and input from the keyboard. Importantly, this will also poll the queueing system designed to manage delayed tasks – this will be treated in the next section.

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.

In case the timer is lagging behind schedule, the latter step 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 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.

4.3 Various deterministic activities

For the moment we shall ignore the activity of players and concentrate on the tasks performed deterministically as time progresses. There are some operations

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

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 queueing tasks and execute then 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.

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 spotting or hiding of units is used by the AI to determine targets.

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

4.4 Player input and network instructions

Suppose the player presses a key or uses the mouse. Either this action regards 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 queued (using the event scheduling framework) until its execution time. 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).

Chapter 5

Networking

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

5.1 Choosing a network model

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

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

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

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

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

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

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

5.2 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.

5.2.1 Interactivity: network instructions

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

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

Let us say that some computer acts as a *server* which keeps track of the frame count, while all players are *clients* connected to the server[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 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.

5.2.2 Synchronization instructions

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

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

5.2.3 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

[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 instructions, which is why we choose this model.

$\mathcal{O}(n^2)$ unless special countermeasures are taken, but real-time strategy games are traditionally played by no more than around 12 players, and with the low per-player bandwidth requirement this remains acceptable.

5.3 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 TCP/IP protocol, and along with the lax latency requirements this shows beyond doubt that TCP/IP is a better choice than UDP (which is generally used for more fast-paced games because it achieves faster response times by sacrificing among other things the guarantee of delivery) 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` routine, 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 the other 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.

- `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 `IOHandlers`.

- `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).

5.3.1 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. 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 5.2.1) 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.

Chapter 6

World of JWars

6.1 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: locations should be represented by pairs of numbers (i.e. only two-dimensional systems are considered), and there should be a way to convert coordinates from any coordinate system to any other that represents the same space.

6.1.1 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 it shall become clear later – we shall use systems of tiles for several purposes, which can only be indexed logically by integers, it is reasonable to consider 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. Assuming that each coordinate represents a small square (and not a rectangle) of real space, two coordinate systems must have the same width:height ratio in order to represent the same space, see Figure **??**.

The drawback of this method 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^{TM}), 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.

6.1.2 List of coordinate systems

blahblah

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 40 pixels.

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

5. Collision detection map. This tiled map serves to localize colliding entities to different subdomains of the world, see section **??**.

6. Vision management map. This is equivalent to the collision detection map, but used for determining whether enemy units are visible, see Section **??**.

7. 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.

6.1.3 Using coordinate systems

6.2 Game data management

This section describes the *data management* strategy used in JWars™. [?] defines a *data-driven* system as "...an architectural design characterized by a 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[1].

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.

6.2.1 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[2] 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. Every type of tank will be distinguished only by data.

[1] 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[?].

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

6.2.2 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 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 unique 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, that is, every configuration of infantry squad and every class of vehicle is represented by a category: 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 **??**.

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, and many of the infantry weapons in the game use the same weapons.

6.2.3 Content loading by categories

Category creation, of course, still requires a lot of data. But only one category is created for every type of unit in the game, and only once, namely when the unit type is first initialized. It therefore makes sense to manage the set of categories in a central *data manager* and repository which the game can use while running.

The JWARS™ data repository stores a dictionary which associates names of unit categories (such as "T-34") with categories (such as the T-34 category). Categories for all units can be accessed through this dictionary, whether they are tanks, infantry units, or even formations such as platoons.

Another dictionary stores the names of weapons and their corresponding categories, and separate dictionaries are used to store tank hull and turret categories[3].

[3] At first sight the use of several dictionaries can be inflexible, since adding new such

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 6.1: *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).*

As promised earlier all this game content is read from external files. The central data manager can conveniently be used to parse datafiles containing unit 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 6.1 and 6.2 which show examples of datafile entries. Notice that many variables are written in terms of metres and seconds. The data manager automatically converts human-readable quantities into the arbitrary system used internally.

When the data manager loads a file, it parses the words (separated by whitespace) in sequence. First it reads the category type identifier ("weapon" or "tank" in the above examples) and uses it fetches the correct category class. Then it invokes the corresponding category constructor which is responsible for parsing the remaining text from a particular datafile entry.

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).

6.2.4 Current game content

6.3 Terrain

terrain: possible effect on movement, hiding, shooting etc

categories would require changing the code of the data manager. Sean Riley[**?**] warns explicitly against this. However in this case, since weapons and units are *vastly* different concepts it is logical to separate them in different dictionaries. In JWars™ all units, whether tanks, infantry or abstract formations such as platoons and companies are stored in the same (unit) dictionary, thus honouring a generic treatment of game objects.

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 6.2: *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 6.1. The other guns and the hull and turret types are also identifiers of categories.*

6.3.1 Representation and capabilities

2D square grid system. Vegetation, possibly details regarding hiding, shooting and movement. Whether or not terrain can be passable (world bounds?).

6.3.2 Terrain generator

Diamond-square algorithm. Buffers, smoothification, etc. Alternative uses of the terrain generator.

6.3.3 Appearance

Randomly generated grass, trees. Rendering by means of images. Several types of each (to make the grid look non-grid-like).

6.4 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.

6.4.1 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 **??**, 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.

6.4.2 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.

6.4.3 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.

6.5 Vision

6.5.1 Vision in games

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 black 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 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), and this possibility should be

included in any realistic wargame[4].

6.5.2 Performance discussion

6.5.3 Final design

[4]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.

Chapter 7

Collision detection

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

7.1 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. Further shall restrict the discussion to two-dimensional collision detection seeing as JWars^{TM} 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.

7.1.1 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

The amount of checks can, however, be reduced by *registering* units in limited subdomains of the world and only checking units in the same subdomain aganst each other (for now assuming that units in different subdomains cannot intersect). Suppose, for example, that the world is split into q parts each

containing $\frac{n}{q}$ units. Then the total amount of checks, being before n^2, will be only

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

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

7.1.2 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–see figure **??**. We shall describe two ways to proceed.

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

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

For the JWARS$^{\text{TM}}$ collision detector we have chosen the second approach, primarily because it does not restrict unit size to any particular scale. This approach will also likely be more efficient since it in most cases will require less than half the number of tiles to be visited (as noted, 4 is a bad case in this model whereas the former model consistently requires 9). However there is one possible problem which is illustrated in figure **??**, namely that two units which

occupy two of the same tiles will (unless carefully optimized out) be checked against each other in each of those tiles[1].

7.1.3 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 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 no custom shape handling will be discussed in this text.

7.2 Design of the collision detector

The collision detector manages a basic kind of entity which we shall refer to as a *collider*. The most basic properties of a collider are its location (x, y) and the radius r of its bounding circle (it has a few more properties which are irrelevant to this section but will be mentioned later). Whether or not a collision has been detected is determined solely by these properties.

7.2.1 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

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

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.

7.2.2 The collision grid

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

7.2.3 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* (cf. the "2.5 D" geometry, section **??**) or other concepts that can desirably be modified.

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

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 **??** 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 in Section **??**. 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.

7.2.4 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 and perform interpolation to ensure

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

7.2.5 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 ??, a tile map of *collision tiles* and a terrain map (Section ??).
- 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.

7.3 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.

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

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 geomtry.

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.

7.4 Pathfinding

For moving units in RTS games the need for a pathfinding algorithm arises. Pathfinders were implemented in the earliest RTS games and have improved through the years. Most pathfinders today extends the normal 'single-source shortest path problem' solution to incorporate unit-to-unit relations which make units capable of interaction for finding the optimal paths. For this project we need a pathfinder to work on the world of JWARS™ while still be a viable solution in similar worlds. With this in mind we will form an algorithm that can be used in other systems as well but using the JWARS™ world as an example.

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

Many of todays RTS games solve this problem by using a tilesystem for the map used for pathfinding and designating tiles with either 'used' or 'free' as markers when scanning through the map with an algoritm[3]. This approach has several advantages, like high and consistent speed, while it requires a predefined map-structure to search in. A good example is the A* algorithm which is a shortest path graph algorithm. For finding a shortesth path using graphs for data representation history has shown that the A* algorithm is viable choice. In any situation we will need a way to represent possible positions of a moving object as fix points so a moveorder can be broken down to to multiple moveorders. The most commonly used approach is the graph representation when solving pathfinding problems. Given a graph represented as follows:

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

V is a list or other representation of all the *vertices* in the graph E is a representation of the *edges* in the graph. An edge is best seen as a link between

[3] 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.

two vertices - meaning that you can go from vertex $v1$ to vertex $v2$ using the edge $e(v1, v2)$. The weight of an edge, corresponding to the amount of time/cost it takes to traverse it, is given by a weight function $w : E \mapsto [0, infinity]$ since a distance already travelled can not be negative.

Given a graph with a chosen data structure there are 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 since this type has the best running times with the fewest vertices visited - like the A* algorithm.

The pathfinding in JWARS^{TM} has some requirements to the algorithm which we must take into account before choosing a final solution. The most pressing issue is to convert the dynamic and rather limitless implementation of units and other objects in the world of JWARS^{TM} see section **??**. We have chosen a very open approach in the area of unit and building location, size and form, which complicates the final form of a pathfinding solution. Any building or unit can be placed anywhere on the map and will not fill out a predefined amount of tiles in the world. The option of letting objects take space on the map, like a chesspiece on a chessboard occupying the field [A,2] can not be used in JWARS^{TM}, since the data structures allows objects of *any* size in JWARS^{TM}. With the predefined restriction in mind we can not use the map alone for writing an effective pathfinder as the amount of information would be lacking. Therefore the most obvious data to use for pathfinding are the actual objects.

If we are to use the object data some rules has to be defined or the amount of different scenarios would become infeasible to comprehend. If the object data is to be used, the most effective way to use them is to treat all objects as convexhulls. Convexhulls has many properties which makes the basics of handling and calculating a lot easier in this project.

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 which employs a heuristic estimation of the distance from any node to the goal. The JWARS^{TM}-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.

7.4.1 Implementation

Though a tile-based system is incapable of handling the pathfinding in JWARS^{TM} - the aspect of pathfinding on graphs is still viable and the most efficient method. The implementation we have chosen for the pathfinding is to transform the dynamic/open implementation of the JWARS^{TM}-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 consist of all corners on static objects - convex hulls - on the map. This data is stored in the collion map. E Is an empty list. [4]

The start and goal location are considered vertices[5] which is specified for each running of the algoritm. In general pathfinding A* is considered the most effective search algorithm on the single source shortest path problem. There exist a number of algorithms to solve the problem but the A* algorithm has the shortest running time and fits or problem profile well in the expansion of the search tree.

In theory no edges are be represented in E. When a node is expanded we get a set of edges based on the current pathfinding problem. This means that everytime we use the pathfinder we have a new setup and all nodes could produce a new set of edges. 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 according to the A* and updates the nodes found by the expand function. [6] 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 not affected by any other part of the game implementation than the collision detector. If a developer wants to use this pathfinder it is fairly easy to convert to a different setup - a conversion need a function which can detect a collision between an 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.

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

The algorithm is started by calling the findPath with a end coordinate and the specified unit. As explained later the pathfinder returns unique solutions to specific specifications. Calling the method with two different sized units can yield two different results. This will be described to depth later in this chapter.

Given the start coordinates as the units current location and the end as argument to the method the standard loop for an A* is implemented. The loop selects a node to expand based on a heuristic evaluation which corresponds to a *priority queue*. The term priority queue will be used throughtout this

[4] 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 alot of memory space.

[5] 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 - this makes us capable of shooting towards and collide with it.

[6] 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

chapter. In a standard implementation it will be referred to as the *openList*. The heuristic evaluation is based on an evaluation in a 2D environment for pathfinding. Taking into account that all distances travelled are straight lines, we can always be sure that we have the shortest possible path to any given node if we use the method normally called 'Relax' as in **??** when describing Dijkstra's algorithm. The g-score for a node 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 possibility to result in getting directly to the node will be the next expanded. This approach mean we can safely terminate the algorithm upon reaching the goal location and have the shortest path possible without further expansion of the algorithm.

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^{TM}. Upon expanding a node we only activate nodes which can be reached in a straight line from the current node. This ensures that all values calculated will distances either already travelled - h value - or the minimum distance - g value - to the goal given that no objects is blocking the line. When expanding a node we expand it towards another node. In JWars^{TM} 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 pathfindingnodes 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 moveables 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 a *indented location*. This location is the dynamic coordinate which will be calcualted in each run through the pathfinder for all relevant nodes. When expanding a node it will always be expanded towards an other node. The pathfinder has a special tileMap called a LineDrawCapableMap. This map is derived from the standard tilemap as explained in **??** and takes the collision map as argument. The LineDrawCapableMap comes with a method which utilises Bresenhaum's line drawing algorithm to find a list of tiles based between two points on the map. This list will consist of CollisionTile's from the collision map and can be expanded to draw a line of certain thickness based on the unit radius. The thickness is calculated as CollisionTile.size / moveable.radius . Using this formula the tiles returned by the function would be the tiles the moveable has a theoritical chance to touch. From the list of collision tiles we acquire a list of terrain objects which should be checked for collision. 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 take more resources. The second step is to calculate all angles to the the pathfindingnodes indented locations in the current object. Calculating the largest and smallest angle we can perform a check wether the line is between these to angles. If we detect a

collision with the object, the rule about all objects being convexhulls gives us the two pathfindingnodes for finding a path around the object. The pseudocode for the expand function show this princip rather well.

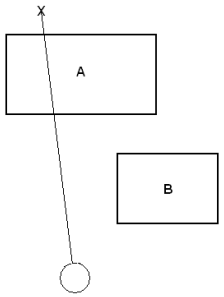
```
Expand(from, to, unit)
min = max = null;
List = getTileList(from, to);
TOList = getTOList( List );

for each object in TOList
{
  if( pointLineDist < unit.radius + building.radius )
  {
    set min, max angles
    if( min != null max != null )
    {
     expand(from, min, unit);
     expand(from, max, unit);
    }
  }
}
```

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

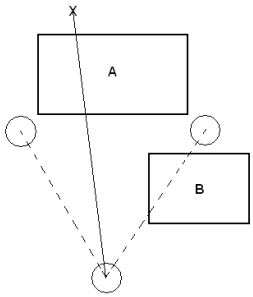
Everytime a position (pathfindingnode) is grey it has been added to the priority queue by the expand function. When the expand function succesfully makes contact with the targetted node we update the target node with the relevant data for the A* algorithm to run as intended. The update method will reevaluate 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. We use values f,g and h. 'f' for the travelled distance to this node, 'g' for the heuristic evaluation to the goal and 'h' as the combined values.

The expand function suffers one fatal error. It can fail in finding all the neccessary edges leaving it. En example of this situation follow here.

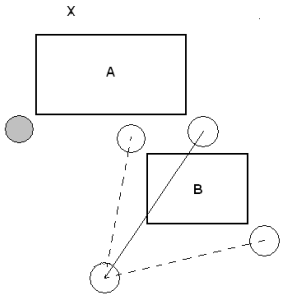
It is clearly that acquiring the nodes on the smaller building would be the fastest route to the target X. The path taking the moveable closer to the object fits a standard tactical manouvre, where covers means safety. In the real world objects on the battlefield would be used by units to hide their positions or make up defenable position. One other error which can be forced by a programmer is create a single structure from multiple convexhulls. We have already stated that in order to have non-flawed data objects must be convexhulls. If a programmer chose to make create a 'U' formed building consisting of 3 rectangles,



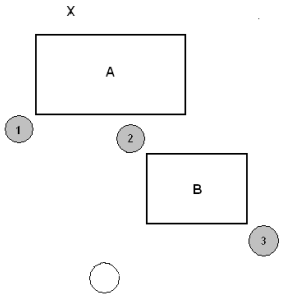
(a) test1



(b) test2



(c) test3



(d) test4

Figure 7.1: *test*

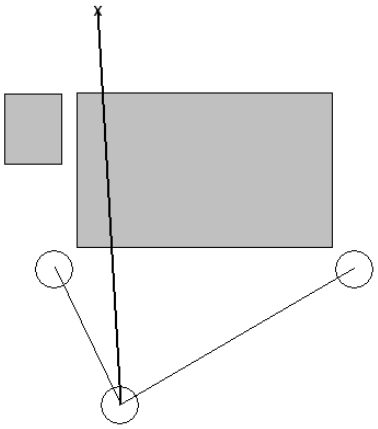


Figure 7.2: *blahblah*

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 foresee other units walk patterns and to take these into their own calculations when finding a route. This possibility do not arise in a world which is not grid-based since the possibilty to 'rent' map space is not available. Unfortunately this option will never be available to a pathfinder not based on the map structuring. In the real world it makes 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 optimally 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.

Chapter 8

Unit organization

The concept of *units* in JWARS™ differs fundamentally from the corresponding concepts in other realtime strategy games. 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™, and this is the most likely feature to make JWARS™ "famous" if such a thing should happen.

8.1 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 rifles 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.

8.2 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 is in 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, these games allow the player to *drag a selection box* on the battlefield 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 around).

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 dependence on 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 dependence on fast mouse action, since the time scales involved in most operations will increase. On the other hand, the reduced dependence 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

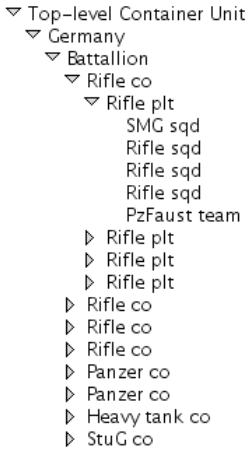


Figure 8.1: *Example of a unit tree. Only the nodes with downward pointing arrowheads are expanded. This is part of a screenshot from* JWARS^{TM}.

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 8.1 shows an example of a military hierarchy in the current version of JWARS$^{\mathrm{TM}}$. This battalion consists of 116 individual entities (vehicles or separate infantry squads), comprising 344 infantrymen and 36 tanks or assault guns.

8.3 Tree-based unit representation

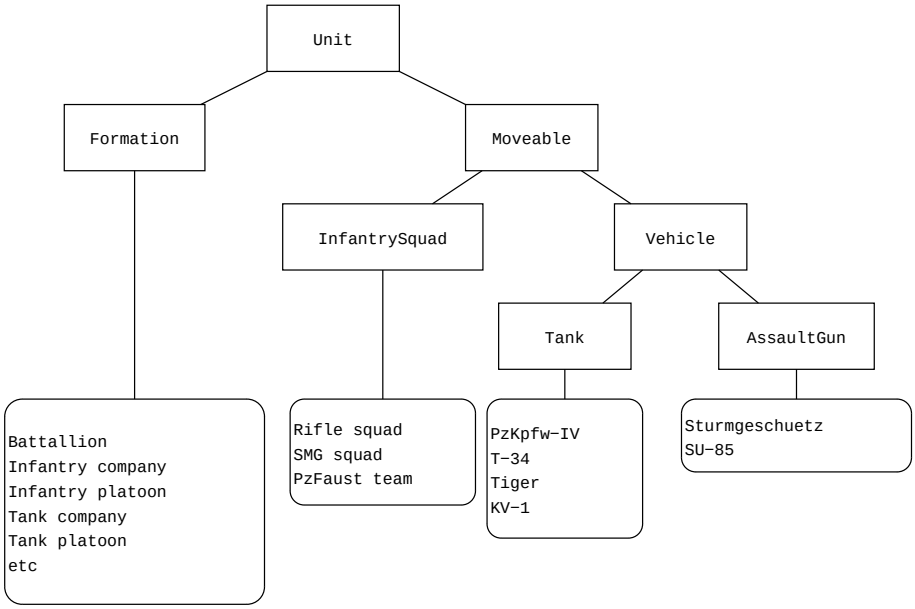


Figure 8.2: *Types of units. The boxes with rounded corners indicate concrete examples of the particular unit type.*

Chapter 9

Unit AI

(mention that AI more or less translates to 'behaviour' in this case)

9.1 Hierarchical structure

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

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

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

- Tactically, if one unit is attacked the entire platoon or company will be able to respond. In classical realtime strategy games this would result in a few units attacking while the rest were standing behind doing nothing.

Thus, this promotes sensible group behaviour which has been lacking in this genre since its birth.

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

- The controls may, as we shall see below, be structured in such a way as to abstract the control from the concrete level in the hierarchy. This means the player needs not bother whether controlling an entire company or a single squad: dispatch of orders to an entire company will invoke the company AI to interpret these orders in terms of platoon operations. Each platoon AI will further interpret these orders and have the individual units carry out the instructions 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[1].

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

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

9.2 Design considerations

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

9.3 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.

9.4 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. blahblah

Chapter 10

Combat

This chapter deals with the combat model provided with JWARS™. The combat model encompasses different modules pertaining to weapons and automatic firing routines, targetting (via the spotting routines of Section ??), armour and damage.

10.1 Analysis of combat dynamics

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.

10.1.1 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*[?] and sometimes *Steel Panthers: World at War*[?].
- 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*[?]), 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[?] notes a particular occasion on which German panzers fired scores of shells at an immobilized Soviet KV-1 heavy tank. Finally the Soviet crewmen emerged to surrender, badly shaken, but unhurt.

10.1.2 "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[?] 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 10.1 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[?].

Thus weapons can 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 other types, they are considered

Figure 10.1: *Soviet T-34 tank with wire mesh for protection against the Panzerfaust anti-tank weapon.[?]*

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.

10.1.3 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 **??**) 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.

10.1.4 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.

If firing is possible, the actual hit location is calculated. 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 in the general area and not on exactly the central point. Other types of 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. All units in this set have

10.1.5 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 vehicle armour model is somewhat more complicated. Vehicle armour is specified by *categories* (see Section **??**). The armour thickness is specified in millimetres, along with the armour plating angle, on the vehicle front, side and rear. Tanks, having a turret, have another such set of numbers, see Figure **??**.

10.2 Spotting and targetting

Chapter 11

Control

Chapter 12

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[?], to scenegraph implementations such as Java3D[?], Xith3D[?] and the game library LWJGL[?]. 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.

12.1 Active versus passive rendering

As mentioned in Section ??, realtime strategy games normally consist 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* which issues repaint instructions continuously. 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.

12.2 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, though it remains important to any rendering system.

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

12.3 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 **??**, 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 12.4.

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*, 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[3].

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^{TM} 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^{TM}, 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 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.

[3]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.

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.

12.4 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.

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 12.2. 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 yields much better performance. Drawing an image (such as the secondary off-screen buffer) is a fast process, whereas the remaining in-game graphics, involving rotations and possibly transparency, are much more time consuming.

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.

12.5 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 13

Conclusion

Bibliography

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

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