Using Interactive Computer Games for AI Research

Case Study on WarCraft III

Master Thesis in Computer Science and Engineering

Jonas Isberg, d97ji@efd.lth.se
Department of Computer Science
Lund Institute of Technology
Lund University
Supervisor: Jacek Malec

June 2004

Abstract

In search for research environments, artificial intelligence has lately taken an interest in interactive computer games. The purpose of this master thesis is to investigate how useful interactive computer games are as environments for AI research. It will be a case study on Blizzard Entertainment's WarCraft III.

The possibilities provided by the environment and the problems faced when creating an AI for a real-time strategy game are studied by the implementation of a primitive AI for WarCraft III.

The study shows that there are several kinds of advanced AI problems involved in creating a dynamic and non-cheating, but yet challenging AI for WarCraft III. It also shows that the WarCraft III environment provides possibilities for solving these problems.

It is possible to study many kinds of AI problems using the WarCraft III engine and creating an AI that plays the standard game-type requires solving many of these problems. WarCraft III can therefore be considered worth testing as an environment for AI research.

Contents

Chapter 1

Introduction

1.1 Background

In search for research environments, artificial intelligence has lately taken an interest in interactive computer games. The environment types which have traditionally been used - military simulations, home-made simulations and robotics - all have their drawbacks: military simulation researchers need to spend a lot of time learning military organization, home-made simulations take time to construct and have been criticized as not being real problems, and when using robotics one has to solve the problem of inaccuracy of sensors and effectors before dealing with cognitive problems. Interactive computer games are less affected by these drawbacks: they take less time to learn than military simulations, they provide an already constructed non-trivial environment, they don't have the problems associated with acting in the real world, but they are complex problems where computers competes against humans, and the computers are often supposed to act human-like.

The benefits of using interactive computer games are that they are cheap, available, come in all kinds of complexities and often have the possibility to create custom game worlds using bundled editors.

There has been some successful research using interactive computer games of the action (or first person shooter) genre, including research on Quake by Laird and his group[1], but there is still little information available on the use and usefulness of interactive computer games in AI research.

1.2 Defining the problem

The rising interest of AI in interactive computer games, the amount of articles[2] which suggests interactive computer games as promising research environments and the lack of available information regarding the usefulness of specific games, suggest that there is a need for case studies on individual games.

Which game (or games) would then be most beneficial to study? Due to the lack of previous studies, it seems a good idea to select a game in which it is possible to study as many kinds of problems as possible. My previous experience of interactive computer games and a few small efforts of creating worlds and AI [1]

[1] AI is used throughout this report both as the name of the scientific discipline and as the

for some of them, have made me believe that Blizzard Entertainment's real-time strategy WarCraft III is a game worth deeper evaluation.

What is it then that makes WarCraft III look promising? WarCraft III is the fourth game of Blizzard in the real-time strategy genre (after WarCraft, WarCraft II and StarCraft), which suggests that they should have been able to create a robust game-engine. It has a world editor with the ability to create custom game worlds and units; and it supports creating agent behavior using both graphical and textual programming.

Thus, it seems useful to make a case study on WarCraft III, regarding its usefulness as an environment for AI research.

1.3 Purpose

The purpose of this study is to investigate how useful WarCraft III is as an environment for research in artificial intelligence.

The focus is put on investigating the possibilities to study multi-agent problems situated in a two-dimensional environment.

1.4 WarCraft III

Blizzard Entertainment's WarCraft III is a real-time strategy game. Strategic and tactical decisions are of highest importance to increase a player's chances for victory. Victory is achieved in the end by crushing the army of the opponent. To make victory achievable, an army must quickly be built, which requires a steady income of resources in the form of gold and lumber. Resources are gathered by workers who mine gold and chop wood.

1.5 Method

This thesis will combine presentation of what can be done using WarCraft III (WC3) with showing how, and why, it can be done. This approach is taken to make the findings of this study as useful as possible and make it as easy as possible for researchers to further evaluate this environment. This means that a part of this thesis will be in the form of a tutorial. This tutorial will show how traditional agent problems can be modeled in WC3 and also will show what problems are involved in creating an AI for a normal game of WC3, as well as suggesting solutions to these problems. These solutions will not be complete, but will show that it is possible to create more advanced solutions for these problems.

---

name of a game module.

Chapter 2

Background

2.1 History of artificial intelligence

Artificial intelligence (AI), uses knowledge and ideas from a number of different sciences. Philosophy has contributed with theories of reasoning and learning. Mathematics with formal theories of logic, probability, decision making and computation. Psychology with tools to investigate the mind. Linguistics with theories of structure and meaning of language. Computer science with tools to make AI a reality.

Artificial intelligence officially got its name 1956, but the first work considered AI was presented 13 years earlier in 1943 by Warren McCullough and Walter Pitts[3]. Artificial intelligence was named on a Dartmouth workshop where a reasoning program, *the Logic Theorist*, was presented by Allen Newell and Herbert Simon[4][5].

The general distrust in the usefulness of computers gave early AI researchers lots of opportunities to show seemingly amazing results. More noteworthy successes includes *the General Problem Solver*, Allen Newell and Herbert Simon[6], and *the Geometry Theorem Solver*, Herbert Gelernter. *LISP* was created 1958 by John McCarthy, the same year as he describes *the Advice Taker* in *Programs with Common Sense*[8], considered as the first complete AI system, although hypothetical.

The concept of *microworlds* were founded, a domain of limited problems which appear to require intelligence to solve, of which the blocks world is the most famous. Works on neural networks, based on McCullough and Pitts, was done and Frank Rosenblatt proved the famous *perceptron convergence theorem*[9].

When researchers tried their skills on harder problems, they came to realize that their previous methods were not sufficient. Even if the problem was solvable in theory, it might take forever to solve it practically.

Knowledge-based systems, or expert systems, came as an answer to the need fore more domain knowledge to solve advanced problems. By coding a lot of domain knowledge into the program, it got a better idea of which steps were useful to try when solving a problem, making them spend less time on *solutions* that obviously are dead-ends.

Expert systems were taken into commercial service in the 1980s. At this time

a Japanese project to build computers that run Prolog programs as machine code was started, and the fear in the U.S. for getting behind Japan made researchers have no trouble finding funding. The 1980s also saw a rise in the interest for neural networks.

The interest in combining the different sub-fields of AI, pursuing one of the original goals of creating intelligent systems, have increased lately. Interactive computer games have been found as interesting environments for AI research by Laird and others.

2.2 History of interactive computer games

The history of computer games is intersected with the history of video games on special purpose home entertainment systems and the history of arcade games.

The history of computer games started 1958 when Tennis for Two was created. It was played on an analog computer, using an oscilloscope for display and is developed by Willy Higinbotham to amuse the visitors of Brookhaven National Laboratory. It was a huge success.

Most of the early computer games were text-based adventure games, with Spacewar being a noteworthy exception. Spacewar was created by Steve Russel on MIT with the purpose of demonstrating hardware abilities.

MUD (Multi-user Dungeon/Dimension) 1979 gave users the possibility to adventure together using a text-only interface.

The text-only interface of the early games was later complemented with rough graphics and with the increased support for graphics and the introduction of 'cheap' home computers, the more graphics-intensive arcade and video games were ported to the computers.

The 1990s saw the rise of the real-time strategy games with Dune II setting the standard, followed by games such as WarCraft, Command and Conquer, Age of Empires, StarCraft and Empire Earth.

The 1990s also saw the rise of the action, or first person shooter, genre with games as Wolfenstein 3D, Doom and Quake (all by iD Software).

The enormous success of *the Sims* has shown that modern people like playing God and controlling other people's life. Predecessors in the God games category includes Black and White and the older Sim City.

2.3 Artificial intelligence and interactive computer games

John Laird has suggested *Interactive computer games as the killer application for human-level AI*[2]. This section will shortly explain why the computer game developers and the AI researchers have so much to gain by cooperating.

Games have long been rated and sold on the number of polygons they are able to produce, but the graphics in most games are now so good that improving it won't add much to game-play. Developers have therefore started to get interested in the possibilities to improve the AI of the computer opponents.

AI researchers on the other hand have started to show interest in computer games, realizing that there exist a lot of ready-made environments which present interesting and complex problems.

2.3.1 Game genres

Artificial intelligence has the potential to play an important role in several game categories. Below is a description of such categories and how AI can contribute to them.

Action games let the user control a character in a virtual environment, in either first or third person view. Achieving the goal often involves lots of violence on enemies. Games in this genre include Descent, Doom and Quake, and AI is needed for computer controlled enemies and partners.

Role-playing games let the user choose between several kinds of characters, such as thieves, warriors and magicians. Games of this genre includes Ultima, Baldur's Gate and Diablo. Some games also give the possibility to play in an on-line persistent world with 1000s of other users. AI is typically used for controlling enemies, partners and support characters like merchants.

Adventure games focus more on puzzle solving than on violence. Early games were text-based, but graphics was introduced when it became available. Game of this genre includes Adventure, Zork and The Curse of Monkey Island. AI is used to control characters that the user needs to interact with to solve the puzzles and continue the adventure.

Strategy games let the user control lots of units, mostly military, to wage war against any enemy. Games of this genre includes WarCraft, StarCraft and Empire Earth. AI is normally used to control the detailed behavior of human controlled units and as strategic opponents. AI could also play an important role in controlling strategic partners and support characters, but this is not yet common.

God games let the user take control of the creation of a city (Sim City), control peoples lives (the Sims) or just play God (Black and White). The AI is used to control the individual units in the game.

Team sports let the user be both the coach of a team and/or take control of a single player at a time. For all somewhat big team sports, there are most likely a computer game. AI is used for individual unit control, for strategic opponents and sometimes even for commentators.

Individual sports let the user perform most of the existing individual sports. AI is used as tactical enemies and sometimes as commentators.

The different genres are getting harder to separate from each other, but the roles that the AI plays are still the same.

2.3.2 Roles

Tactical enemies need to make quick decisions in battle situations, no matter whether they are military tanks or hockey players.

Partners are similar to tactical enemies, but where it is important for a tactical enemy to operate autonomously, a partner should focus on frictionless cooperation.

Support characters are usually very simple and the interface for communicating with them primitive. There are many ways these could be further developed and they have the potential to dramatically [improve] gameplay.

Story directors adapt the evolving story according to the action of the human player. The intent is to make sure that the player does not miss any interesting events; but that events happen when the player is around, or happen where the player is located.

Strategic opponents have traditionally resorted to cheating, by having full knowledge of the location and strength of the enemy and/or having more and better units to command.

Units are often given high-level commands which the unit AI carries out. One important issue is to know when to (temporarily) disobey a command. When an enemy unit is spotted, it might be better to respond fire, if being attacked, than to continue with the previous activity.

Commentators analyze and commentate the game situation as the game progresses.

2.4 Real-time strategy games and WarCraft III

A standard RTS game involves several players fighting each other to death using armies. The players can be a part of a team or fight alone, and the players can either be controlled by a human or by the computer. The part of the AI that controls the computer-controlled player's high-level actions are considered a strategic opponent. The AI part that controls the detailed actions of every single soldier, no matter if they belong to a computer or human controlled player, are the unit AI.

Most RTS games include a few different game-types: melee, campaign and custom. A melee game follows all the default rules and can be considered as the standard game-type. Each player normally starts with a few workers and a main building. The goal is to crush all enemies. A campaign can be more or less modified from the melee game. It is normally more story based and often contains quests which need to be fulfilled. It does also normally stretch over several maps/levels/worlds. Those which do not fit into the previous categories are considered as custom games. Their game-play can be vastly different from each other. Some popular custom game types are tower defenses and arena games.

Chapter 3

Experiments with the game environment

This chapter presents how game worlds and AI agents can be created for WarCraft III. It starts with looking at the tools available for creating game worlds and AI agents, including both tools that come bundled with the game and other tools created by users. Thereafter the Jass scripting language is introduced, which is used for creating advanced functionality that is not directly supported by the graphical tools. When all the concepts and tools have been introduced, it is shown how agents can be implemented. It first presents how simple agents which are run separated from the world and only communicate with it by a formal percepts action interface, can be created. Thereafter the problem of creating an AI player for the a normal game of WarCraft is described. Finally it is shown how other game types can be created using WarCraft III.

3.1 Introduction to the World Editor

Bundled with WarCraft III comes the World Editor (WE), a tool with which maps can be created. The WE consists of several individual tools, including tools to modify the terrain, create the (physical) laws of the world, import custom made objects, customize objects, create campaigns that include several maps, support for creating AI, etc.

The Terrain Editor

: The Terrain Editor is used for modifying the ground of environments (maps). The ground can be set to different terrain types, like grass, plains or dirt. The ground can also be set to different heights. This is also the tool with which one can pre-place buildings, units and items in the world, as well as define regions (for later use). The Terrain Editor is normally used for creating the static parts of the world and sometimes for placing units at their initial positions.

  When the WE is started, it is actually the terrain editor that starts, containing commands to access the other tools.

The Trigger Editor

The Trigger Editor is the tool for creating game dynamics (the laws of the world). Triggers can be created graphically, built on the event- condition-action concept. If an event that a trigger has registered on occurs, then the triggers condition is checked. If the condition is true, then the action is run. It is also possible to create triggers using textual programming in the Jass language (see section about Jass). Most part of the static game world which is set up using the terrain editor can also be done at game startup using triggers.

The Sound Editor

The Sound Editor is used for all handling of sound.

The Object Editor

The Object Editor can be used to customize objects. Changing the properties of units, like strength, intelligence and speed, can be done here.

The Campaign Editor

The Campaign Editor is used for creating campaigns that stretch over several maps.

The AI Editor

The AI Editor is a graphical tool that can be used to create a simple AI. The AI can then be exported to Jass and further modified textually. Although the behavior of the AI created with the AI Editor is not very flexible, it is a good way to prepare oneself for more advanced AI coding in Jass.

The Object Manager

The Object Manager gives an overview of all objects in the currently viewed map. It lists units, buildings, items, triggers etc.

The Import Manager

The Import Manager is the tool used for importing custom made objects and files into a map. Things that can be imported this way include, custom AI files, characters, etc. The most important use of the import editor while creating an AI would be the ability to import AI scripts into maps. (Since this is only available in the *WarCraft III: The Frozen Throne Expansion*, one has to use third party tools, mpq readers, to get this functionality when using WarCraft without the expansion.)

3.2 Introduction to the Jass scripting language

Jass is the language that the map files are actually saved in. Jass is very flexible compared to the graphical tools included in the WE, but compared to other programming languages it is rather simple.

3.2.1 Jass syntax

This section is a short introduction to the syntax of Jass. It focuses mainly on things that are different in Jass compared to languages such as C, Java and Perl. More details about the syntax of Jass can be found at http://jass.source.net. Jass has the usual programming features like conditions, loops and functions. All the action starts in the main function, although main is sometimes generated by the tools.

main

Just like in a normal C program, all execution of Jass code is started in the main function. The only time one, normally, has to write this oneself is in AI scripts though (more on this later). Think of *nothing* as being equal to *void*.

```
function main takes nothing returns nothing
  code
endfunction
```

set

Assignments in Jass are forced to use the set keyword.

```
set variable = value
```

loop

The support for loops are limited to the *loop* construct, which compared to *while* and *for* loops is quite low-level, although flexible. It is possible to have as many exit-conditions as needed.

```
loop
  code
  exitwhen condition1
  code
  exitwhen condition2
  code
endloop
```

if

If constructs follow the normal semantics. Observe the use of the elseif keyword, when nesting ifs.

```
if condition then
  code
endif

if condition then
  code
else
  code
endif

if condition1 then
```

```
  code
elseif condition2
  code
else
  code
endif
```

call

All function calls that are not part of an expression must start with the call keyword.

```
call someFunction(arg1, arg2...)
```

function

Jass functions are defined the following way:

```
function takes integer a, string b returns nothing
  code
endfunction
```

globals

All global variables needs to be placed in a *globals* block at the beginning of the file. Inside the *globals* block, variables are declared the same way as local variables are in functions. When creating map scripts (triggers), global variables are created with a graphical interface.

```
globals
  variable definitions
endglobals
```

local

```
function main takes nothing returns nothing
local integer number
local integer anotherNumber
local float decimalNumber = 3.14
local string someText

  set number = 42
  set anotherNumber = 6 * 9
  set someText = "local variables are first in functions."
endfunction
```

3.2.2 Scripts

Map scripts are programs that are created using the World Editor, by either graphical or textual programming. When starting a game, it is actually a map script that is started. The map script then, sometimes, starts a number of AI scripts, maximum one for each computer player maximum. There are some things that can only be done in map scripts, and not in AI scripts. One such thing is triggers, which are run when specific events happen during the game.

AI scripts are used for controlling individual computer players. One script is run for each of the computer players (the same script can be used for several players though). A minimal AI script (that does nothing AI-related) could look like:

```
function main takes nothing returns nothing
  call DisplayTextToPlayer(GetLocalPlayer(),0.0,0.0,"Hello, World!")
endfunction
```

3.2.3 The native libraries

There is a lot of native code in the game-engine which can be called using an interface which is defined in the files: common.j, common.ai and Blizzard.j. These files contain both functions implemented in Jass and declarations of functions that are natively implemented.

common.j

Functions that are usable from both AI scripts and map scripts are declared in common.j. There is an issue though with functions that returns strings. These functions do not currently work properly when called from AI scripts. The same goes for most functions that take code/functions as parameters. The functions below are quite useful for debugging purposes.

```
constant native GetLocalPlayer takes nothing returns player
    local Player p = GetLocalPlayer() // The user controlled player.
```

Can be used together with DisplayTextToPlayer (see below).

```
native DisplayTextToPlayer takes player toPlayer, real x, real y,
    string message returns nothing}
    call DisplayTextToPlayer( GetLocalPlayer(), 0.0, 0.0, "Hello local player!")
```

Displays a message to the specified player, the local user in this case.

```
native DisplayTimedTextToPlayer takes player toPlayer, real x, real
    y, real duration, string message returns nothing
    call DisplayTimedTextToPlayer( GetLocalPlayer(), 0.0, 0.0, 60, "60 secs!" )
```

The same as DisplayTextToPlayer, but now we control how long the text shall be shown.

common.ai

Functions that are only usable from AI scripts are declared in common.ai. Most functions in this file are specifically aimed at creating an AI that plays game similar to the standard melee game. But there are also a few which are flexible enough to be useful on most kinds of maps. Below follows some functions usable for gathering resources and constructing units and buildings.

```
native Sleep takes real seconds returns nothing
    call Sleep( 3.5 ) // Will make an AI script sleep 3.5 seconds.
```

The game engine will kill the AI script if it does to much work with no sleeping.

```
native ClearHarvestAI takes nothing returns nothing
    call ClearHarvestAI() // Orders the harvest AI to stop harvesting.
```

This is used with the two functions below to control how many workers that harvest gold and lumber.

```
native HarvestGold takes integer town, integer peons returns nothing
    call HarvestGold( 0, 5 )
```

The above call tells 5 workers to start harvesting gold in town 0, which is the starting town. If this same call is made twice, without a ClearHarvestAI, 10 workers instead of 5 will harvest gold.

```
native HarvestWood takes integer town, integer peons returns nothing
    call HarvestWood( 0, 10 )
```

Works exactly as HarvestGold, but for wood instead.

```
native SetProduce takes integer qty, integer id, integer town returns boolean
    local boolean success = SetProduce( 1, 'hpea', 0 )
    call SetProduce( 1, 'hpea', 0 )  // Ignores the return value
```

Tries to produce one peasant at town 0 (the starting town).

```
native GetUnitCount takes integer unitid returns integer
    set numberOfPeasants = GetUnitCount( 'hpea' )
```

Count the number of peasants the AI player has (executed in AI script). This value includes those currently in training.

```
native GetUnitCountDone takes integer unitid returns integer
    set numberOfPeasants = GetUnitCount( 'hpea' )
```

Count the number of peasants the AI player has. This value does not include those in training.

Blizzard.j

`Blizzard.j` contains files that are only usable in a map script. Many of these functions wraps functions in common.j, with the purpose of making them more compatible with the graphical programming interface. When converting triggers done graphically to Jass, most functions in the code are from Blizzard.j. For a programmer, the functions in common.j are often more intuitive though.

3.2.4 Limitations and solutions

Jass has a few built in limitations. These includes: not being able to pass arrays as arguments or return values, no support for more advanced data types like structs. One solution to these problems would be to use global arrays as memory areas and use integers to index into these areas. Another way to avoid many limitations are to use the game cache. This is an interface to store primitive data types into some kind of natively implemented hash table, using strings as keys.

3.2.5 Useful tools and resources

There are not any official support from Blizzard for developing maps and AI for WarCraft III. Communities and resources made by users are therefore important sources of information.

Jass documentation The most complete jass documentation can be found at `http://jass.sourceforge.net/`.

Jass and AI forum There are several community sites created for those interested WarCraft III modding. A good place to start: `http://www.wc3campaigns.com/forumdisplay.php?f=65`.

pjass Jass syntax checker, useful for checking AI scripts. Downloadable from `http://jass.sourceforge.net/`.

AMAI Advanced Melee AI, configurable AI that plays WarCraft III melee games (see `http://amai.wc3campaigns.com/`).

ejass Jass Preprocessor. Useful for translating resource files into pure Jass code. Included in AMAI download bundle.

3.3 Creating simple agents using WarCraft III: The Wumpus world

As an introduction to the process of creating worlds and agents for the WarCraft II engine, the Wumpus world[10] has been implemented. It is shown how worlds (maps) can be created with a combination of textual and graphical programming and how triggers can be made to interact with AI scripts provided in external files. The intent of this section is both to show that well known problems can be implemented in WarCraft III as well as showing how it can be done.

3.3.1 Overview of the Wumpus world

The Wumpus world is divided in squares surrounded by walls. Each square might contain one or several of the following: deadly pit, the Wumpus monster, chest of gold, an adventurer searching for gold, stench from the Wumpus, breeze from nearby pit. The interesting agent problem is the following: the adventurer should search for the gold, pick it up, return to the entrance, located at (1,1), and climb out with the gold. While doing this, he must avoid the Wumpus, who kills anyone entering the same square as it (there is a stench on all four squares that lie nearby the Wumpus) and avoid the pits (there is a breeze in the four surrounding squares).

The agent possibly perceives the following of its surroundings: Stench from nearby Wumpus, Breeze from nearby pit, Bump from walking into a wall, Glitter from gold in the same square, Scream from a dying Wumpus. The agent can in each turn do one of the following: turn 90 right, turn 90 left, walk forward, grab gold, shoot arrow (he only has one), climb out (if he is at the entrance at (1,1)).

Both the Wumpus and the gold start at a location unknown to the agent. The agent starts at the entrance at (1,1). In each square there is a possibility

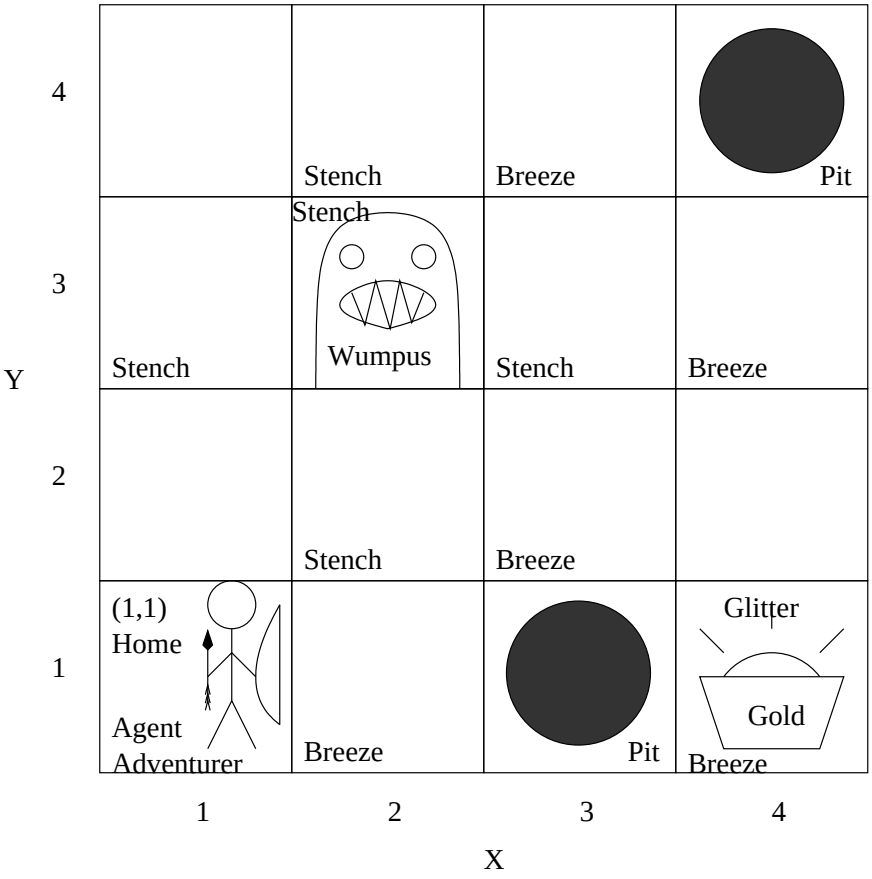

Figure 3.1: A 4x4 Wumpus world, with one adventurer, one Wumpus, two pits and one chest of gold.

that there is a pit (the Wumpus is to big to fall into the pit though). The agent dies if he enters a square with a pit or a live Wumpus. The Wumpus dies if the agent manages to shoot it with his arrow.

3.3.2 Design

To minimize the trouble of switching the agent AI module, the agent is implemented in a separate AI script file. The world feeds the agent with percepts using the feature for sending AI commands from the map script to AI scripts. This feature does not have any support for the agent in the AI script to send back any information though. This is instead handled by the possibility to assign an integer value to each unit. The agent AI assigns an integer value to the unit representing the agent, which specifies which action it has chosen. The world polls this value, updates the world state and sends new percepts to the agent.

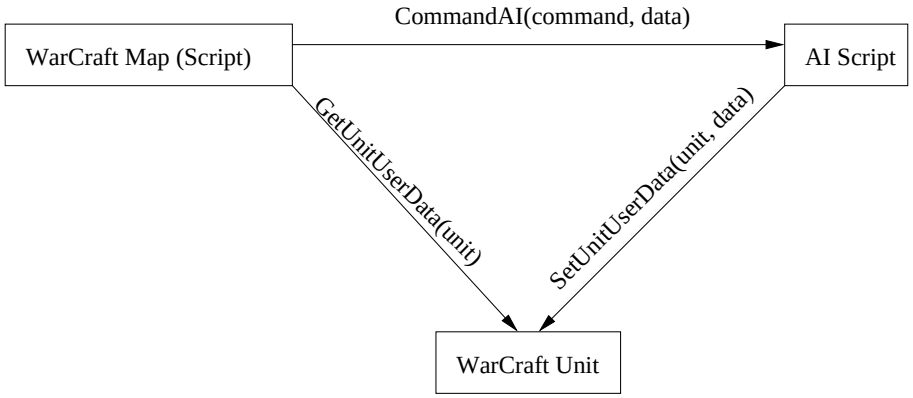

Figure 3.2: Design overview of Wumpus world.

The world, in this implementation, consists of 4x4 squares surrounded by a wall, in the form of farms. Each square is the size of a farm (which in WarCraft III world coordinates are 128x128). The farms are created programmatically using the Trigger Editor. The location of the gold is represented by two global integers created using the Variable Editor and is represented visually by an item looking like a chest of gold. The location of the Wumpus is represented the same way, but will visually be represented by a monster unit. The same goes for our hero, the agent, but it will visually be a farmer. The pits are represented by a 4x4 map coded as a boolean array of 16 positions. The lack of visual pits forces the visual representation of the pit to be a drake (the breeze coming from the flapping of wings).

The agent AI is implemented in an AI script file. It tries to collect the gold, return to entrance and climb out. The agent is controlled by player 2, which can accessed with a call to Player(1) (In GUI the numbers are 1 based, but they are 0 based in Jass).

This implementation encodes the percept sequence in the data field of the CommandAI call. The percepts are coded as:

```
Stench = 1
Breeze = 2
Glitter = 4
Bump = 8
Scream = 16
```

After having retrieved and decoded the percepts the agent chooses one of the actions, which are encoded as follows:

```
Walk = 1
Left = 2
Right = 3
Shoot = 4
Grab = 5
Climb = 6
```

Another way needs to be found to send the chosen action back (to the map/world script), because the CommandAI only supports map to AI script communication. There are a few different ways to do this, one of them is to use the possibility to store an integer associated with each unit, another is to use a player's amount of gold and lumber. This implementation uses the unit that represent the agent, for communicating the chosen action back to the map script. The map script then polls this value periodically and updates the world state according to the chosen action. The integer associated with the unit can be manipulated using calls to :

```
call SetUnitUserData( theUnit, theInteger )
set theInteger = GetUnitUserData( theUnit )
```

The question is then how to get hold of theUnit: the code below achieves this (if the agent is represented by a night elf *archer*):

```
function getAgent takes nothing returns unit
  local group grp = CreateGroup()
  local unit u = null

  call GroupEnumUnitsOfType(grp, "archer", null)
  set u = FirstOfGroup(grp)
  call DestroyGroup(grp)
  return u
endfunction
```

3.3.3 Implementation

The Map Script

The map scrip uses two triggers, one for initializing the world at startup, and another for periodically sending percepts and receiving actions from the agent.

The initializing trigger looks graphically [1] like:

[1] When having been created using graphical programming.

```
Initialize
  Events
    Map initialization
  Conditions
  Actions
    Custom script:  call initialize()
```

Converting it to custom text using the Edit menu gives us the code:

```
function Trig_Initialize_Actions takes nothing returns nothing
  call initialize()
endfunction

//===========================================================================
function InitTrig_Initialize takes nothing returns nothing
  set gg_trg_Initialize = CreateTrigger(  )
  call TriggerAddAction(gg_trg_Initialize,function Trig_Initialize_Actions)
endfunction
```

Since we only want the call to `initialize()` to be executed at startup, and therefore only want the trigger Initialize to be executed once (at startup), the code could have been written as follows:

```
function InitTrig_Initialize takes nothing returns nothing
  call initialize()
endfunction
```

The second trigger is the one that runs periodically and are used to communicate with the AI script. This looks graphically like:

```
Communicate
  Events
    Time - Every 5.00 seconds of game time
  Conditions
  Actions
    Custom script:  call communicate()
```

Converting it to custom text yields:

```
function Trig_Communicate_Actions takes nothing returns nothing
  call communicate()
endfunction

//===========================================================================
function InitTrig_Communicate takes nothing returns nothing
  set gg_trg_Communicate = CreateTrigger(  )
  call TriggerRegisterTimerEventPeriodic( gg_trg_Communicate, 5.00 )
  call TriggerAddAction(gg_trg_Communicate,function Trig_Communicate_Actions)
endfunction
```

The implementation of `communicate()` and `initialize()` is done in the area for Custom Script Code, which is accessed by opening the Trigger Editor and selecting the map name, on the top left, just below the button bar.

The above code requires the WarCraft III expansion (The Frozen Throne), because the original WarCraft III does not support putting code in the custom script area in the World Editor. See appendix B for a full implementation of the map script (one that does not require the expansion). The benefit with having the expansion is that it is easier to manage the code. Functions defined in one trigger can not be used in another trigger, and functions used in several triggers are therefore best placed in the custom script area.

The AI script

The AI script repeats reading percepts, choosing action and sending action, in an eternal loop.

```
function main takes nothing returns nothing
  local integer action
  local integer percepts

  loop
    set percepts = readPercepts()
    set action = chooseAction(percepts)
    call sendAction(action)
  endloop
endfunction
```

The percepts that are sent from the map script with a call to CommandAI are read by readPercepts in the following way:

```
function readPercepts takes nothing returns integer
  local integer cmd
  local integer percepts

  loop
    exitwhen CommandsWaiting() > 0
    call Sleep(0.5)
  endloop

  set cmd = GetLastCommand() // Ignored for now...
  set percepts = GetLastData()
  call PopLastCommand()

  return percepts
endfunction
```

and the chosen action is sent back to the map script with this code:

```
function sendAction takes integer action returns boolean
  local unit u = getAgent()

  if u == null then
    return false
  else
    call SetUnitUserData(u, action)
    return true
  endif
endfunction
```

The implementation of `getAgent()` in the above code has been described earlier in this section. The only thing missing now is the implementation of `chooseAction()`, which contains the implementation of the agent for the Wumpus world.

```
function chooseAction takes integer percepts returns integer
  local integer action

  call decodePercepts(percepts)

  if isGlitter() then
    set action = GRAB
  elseif isBump() then
    set action = TURN_LEFT
  else
    set action = WALK
  endif

  return action
endfunction
```

See appendix B for a complete implementation of a somewhat more advanced agent.

3.4 Building an AI that plays WarCraft III

This section will discuss how an AI agent that plays a normal game of WarCraft can be built. The design is highly inspired by Bob Scott[11]. It will start with a description of normal game of WarCraft III and the problems which both a human and a computer player face. A discussion of how a computer player can be split into different cooperating sections then follows, as well as individual sections describing these parts more in detail and what problems they try to solve. Finally it is discussed how these parts can be made to work together to build a non-cheating AI player.

3.4.1 A standard game of WarCraft

In a normal (melee) game each player starts with one main building and a few (5 for most races) workers and the goal is to destroy the buildings and units of the opponents. To do this, one needs to quickly gather resources, scout the map to see what the opponents are up to and finally build an army with which one is able to beat the opponents. In short, one has to consider the following activities: resource gathering, training of units, construction of buildings, research of upgrades and abilities, scouting and combat, and finally how all these things should be done in relation to each other in an overall strategy.

3.4.2 The resource manager

The first task involved in creating an army is to quickly gather resources. Resources are gathered by workers and are in the form of gold and wood (lumber).

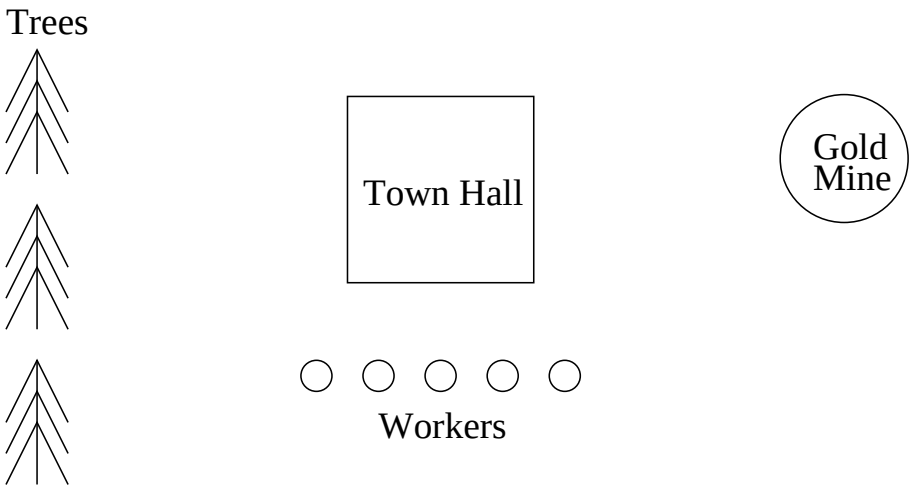

Figure 3.3: Human starting base at the start of a melee game.

The most basic task when gathering resources is to decide how many workers should be given the task of gathering each of the resources. It would also be good if the resource manager could estimate how quickly resources can be gathered and detect when it needs more workers.

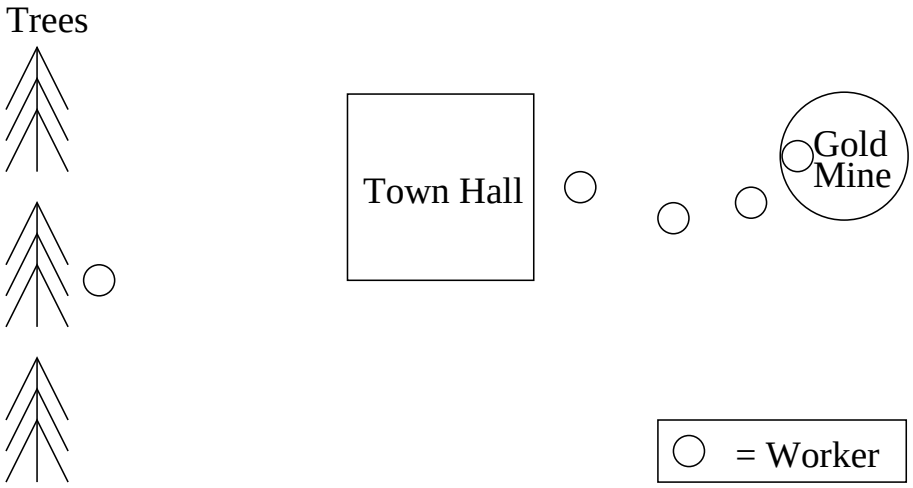

Figure 3.4: Human base with four workers mining gold and one worker chopping wood.

Let us assume that the task of gathering resources is delegated to a resource manager. What does this manager need to perceive of the world state and what commands and questions would a more strategic part of the computer player need to give the resource manager? A very simple, but still relatively useful, resource managing agent could be built which takes no percepts, is updated

every few seconds and gives as output the order to put a handful of workers to work in a mine, and the rest to chop wood. To illustrate what interaction with the game-engine is needed to actually implement this behavior, it is implemented in the following AI script.

```
function resourceManager takes nothing returns nothing
  local integer goldPeasants = 4
  local integer woodPeasants = GetUnitCountDone('hpea') - goldPeasants

  call ClearHarvestAI() // Reset the built in harvest-manager and then
  call HarvestGold( -1, goldPeasants ) // assign miners
  call HarvestWood( -1, woodPeasants)  // and wood choppers.
endfunction

function main takes nothing returns nothing
  call Sleep( 1.0 )  // Sleep 1 second
  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script started")

  loop   // start eternal loop
    call resourceManager()
    call Sleep( 1.0 ) // The engine requires us to sleep ones in a while.
  endloop

  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script exiting")
endfunction
```

3.4.3 The unit manager

When the initial workers are sent off to gather resources, it is time to train more workers to be able to increase the speed of the resource gathering even more. Workers are trained at the main building, the building which every player start with. This building can train one unit at a time, but can be given order to train several units, one after another. Let us assume that resource gathering is the only thing of interest right now, then it would seem reasonable to train as many workers as possible and a first simple unit manager, responsible for the training of units could be given the behavior of training as many workers as possible, as quickly as possible. For each time the unit manager runs, it orders the main building to start produce one worker (if the main building already is training a worker, it will simple ignore the order).

```
function unitManager takes nothing returns nothing
  call setProduce(1, 'hpea', -1) // Produce one peasant anywhere.
endfunction
```

This simple unit manager will help to increase the resource gathering a bit, but is very limited. It doesn't have any idea how to train something other than worker (peasant), and it will try to do so even when there is no chance of success. It would be a good idea if it knew which units it was able to produce and it would also be good if it knew which units it would be a good idea to produce. Before it can produce more than a handful of workers, farms need to be build to produce food for the workers (as well as other units). There is also a need for different buildings to be able to train other types of units.

3.4.4 The building manager

The building manager's task is to construct all necessary buildings and to know which buildings depend on other buildings already existing. The first problem the building manager is faced with is to create farms so that more workers can be built. Every farm provides food for 6 workers (and the main-building for 12). After the simple unit manager has created 7 workers (in addition to the 5 from the start), his attempts at building more workers will fail until either a worker dies, or more farms are built. Two possible solutions to this problem would be either that the unit manager tells the building manager that more farms are needed or that the building manager realizes it by itself. The simpler solution would be that the building manager just builds as many buildings as it thinks is needed. This could be done the following way.

```
function buildingManager takes nothing returns nothing
  local integer produced = 12 + 6 * GetUnitCount('hhou')
  local integer used = 1 * GetUnitCount('hpea')
  local integer foodSpace = produced - used

  if foodSpace < 6 then
    call DisplayTimedTextToPlayer(GetLocalPlayer(),0,0,"Building house")
    call SetProduce(1,'hhou',-1)
  endif
endfunction
```

This simple solution together with the previous solutions for the resource and unit managers would, if they where put together, produce a horde of workers, of which a few would mine gold and the rest chop wood. The workers would also obey the build orders from the building manager when needed (one worker is needed to construct a building). Such a simple main function could look as follows:

```
function main takes nothing returns nothing
  loop
    call resourceManager()
    call unitManager()
    call buildingManager()
    call Sleep(2.0) // Yield to other thread (or get killed by engine).
  endloop
endfunction
```

These simple managers would together form a computer player that is rather good on one thing only, collecting resources. It would be stupefied when the nearby mine runs out of gold, the nearby trees are all chopped down, or an enemy approaches. It is also obvious that these managers should be forced to cooperate more actively with each other and that they need much more advanced behavior. They are also not very similar to formal agents, because they don't take any percepts as input and do not produce any action, but they rather just question and manipulates the world state on their own. But before considering how these primitive managers can be extended, it would probably be a good idea to consider what overall task are they really trying to solve. The overall task could be to win the game by crushing the enemy, or it could be to be as entertaining as possible to play against. But let us assume that its task is to win against the opponent while as much as possible avoiding to cheat.

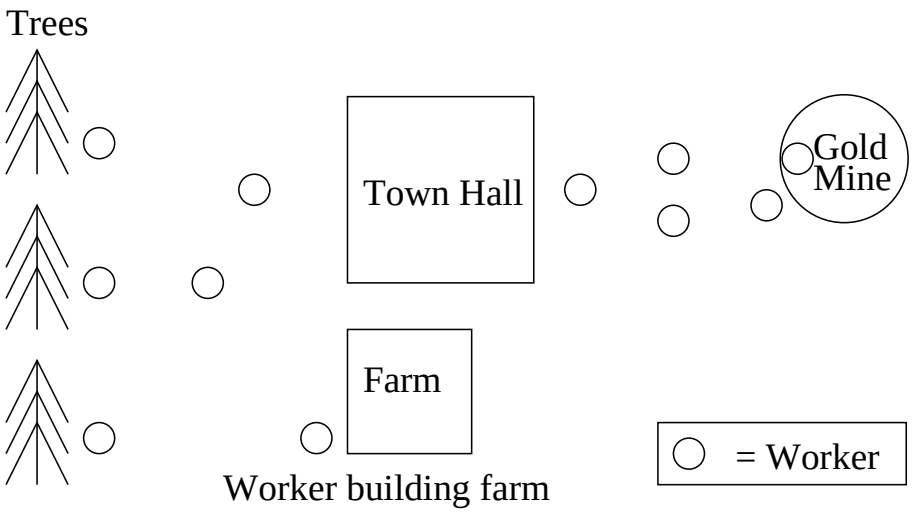

Figure 3.5: Human base with workers gathering resources and one worker constructing a farm to make the training of more units possible.

3.4.5 Training an army

On of the more important sub-tasks of winning against the opponent is to produce an army which is powerful enough to beat the army of an opponent. The tasks which are involved in creating this army are: resource gathering, unit training and building construction, which have been briefly discussed, as well as researching upgrades for different military units. To be able to gather the resource needed to train a powerful enough army, there is often a need for expansion. There can only be a limited number of workers efficiently mining a mine at the same time, and therefore it is advisable to expand to other mines when the opportunity is available. This expansion needs support from military units to secure the new area and to prevent monsters and opponents from attacking the workers. It is not possible to first create a large army and then, when complete, to start fighting the opponent. All available military units need to protect workers and buildings as well as scout and aid expansion from the minute they are completely trained. The code below implements an AI for a computer player of the human race which created 12 workers and a horde of foot soldiers. The soldier will not act unless they are provoked, though.

```
function resourceManager takes nothing returns nothing
  local integer goldPeasants = 5
  local integer peasantsDone = GetUnitCountDone('hpea')

  if peasantsDone < 6 then
    set goldPeasants = peasantsDone - 1
  endif

  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "In manager")
  call ClearHarvestAI()
  call HarvestGold( 0, goldPeasants )
```

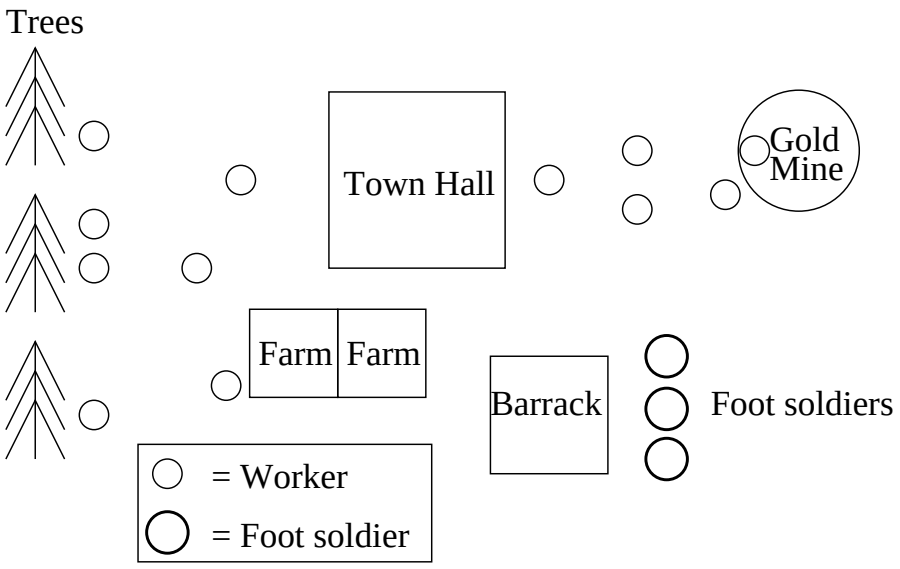

Figure 3.6: A barrack is training foot soldiers.

```
  call HarvestWood( 0, GetUnitCountDone('hpea') - goldPeasants )
endfunction

function unitManager takes nothing returns nothing
  if GetUnitCount('hpea') < 12 then
    call SetProduce(1, 'hpea', -1) // Train one peasant anywhere possible
  endif
  call SetProduce(1, 'hfoo', -1) // Train one foot soldier anywhere possible
endfunction

function buildingManager takes nothing returns nothing
  local integer produced = 12 + 6 * GetUnitCount('hhou')
  local integer used = 1 * GetUnitCount('hpea') + 2 * GetUnitCount('hfoo')
  local integer foodSpace = produced - used
  local integer barrack = GetUnitCount('hbar')

  if foodSpace < 6 then
    call DisplayTextToPlayer(GetLocalPlayer(),0.0,0.0,"Building house")
    call SetProduce(1,'hhou',-1)
  endif

  if barrack < 1 then
    call SetProduce(1,'hbar',-1)
  endif
endfunction

function main takes nothing returns nothing
  call Sleep( 1.0 )  // Sleep 1 second
  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script started")
```

```
  loop   // start eternal loop
    call resourceManager()
    call unitManager()
    call buildingManager()

    call Sleep( 1.0 ) // Yield to other threads and processes.
  endloop

  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script exiting")
endfunction
```

Involved in creating an effective army is also the need to have knowledge about which units fight well together and which units are effective against the units available to the enemy. To be able to reason about which buildings are needed to create the wanted units, how much it will cost, and how long it will take, very good knowledge about the technology tree is needed. The technology tree decides which dependencies exist between different units and buildings. The high-level reasoning would probably benefit from being separated out from the more low-level tasks of constructing individual units and buildings. In this work we put the strategical reasoning in the civilization manager.

3.4.6 The civilization manager

The civilization manager has the role of organizing all the other parts of the computer player. It deals with the strategic reasoning and strategic decisions and queries and gives orders to other parts of the system. The main function in the previous code example can be seen as a very primitive civilization manager, it just tells the other parts to do what they see fit. A handy feature during development would be if the civilization manager could be controlled by the user. As we have seen in the section "Creating Simple Agents with WarCraft III", the only data that easily can be handed to an AI script is an integer. This is not a huge problem though, because most things are actually coded as integers, although integers can be written in four character representation as well ('hpea' is an integer). Which commands should then the civilization manager accept? Considering the current functionality of the other managers the following commands seems appropriate:

```
Harvest(gold, wood)
Train(N, unit)
Build(building)
```

There could be a question of what these commands really mean though. Is the harvest command about how much gold and lumber the computer player should have in total or how much more it should harvest, independent of the total? Should we have N units in total or just produce N, not considering the total at all? It is probably good to have both versions of these commands.

See appendix D for an implementation of a command parser that could be extended for controlling a civilization manager or to communicate with any AI script.

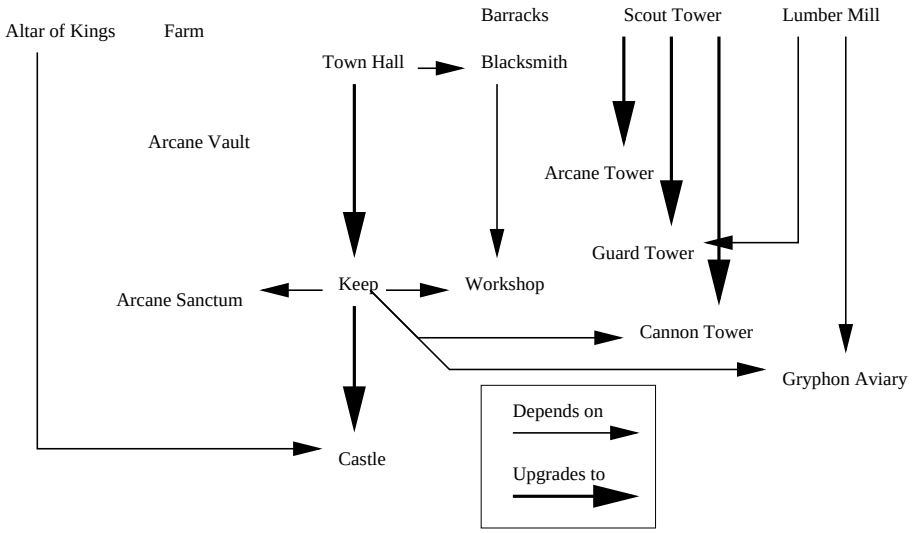

Figure 3.7: Overview of the technology tree for the human race. See `http://www.battle.net/war3/human/buildings.shtml` for more details.

3.4.7 The technology tree

The technology tree contains all information about buildings, units, upgrades and abilities, as well as how they relate to each other. There is unfortunately no known way of getting all this information from the game during runtime. Some of the information is available, but far from all. The needed information has therefore to be coded into the source code of the computer opponent. This might seem like terribly much work to do, but by using code-generating third party tools and using files containing raw technology tree data provided by others, it is quite manageable. The work of both the unit manager and the building manager would be a lot easier if they could query some separate part about dependencies and data, and not having to rely on having this information hard-coded.

3.4.8 The combat manager

Now it is time to consider one of the hardest problems one is faced with when creating a computer player for an RTS game, i.e., Commanding troops, outside and inside a battle. Creating a fully operational and powerful non-cheating combat manager is way out of scope of this study.[2] This study instead investigates how the built-in AI controls its troops, and what alternatives there are to the cheating it resides on regarding choosing attack targets.

These alternatives include the use of simulated computer vision, map-making and path-finding. The normal way of the built-in AI includes asking the engine for interesting attack targets, considering how dangerous they are and if the opponent is protecting them. This way, the AI can, without having been in contact with its enemy, know when the enemy is expanding, and directly initiate

[2]See AMAI[12] for an operational AI that tries to avoid cheating.

an attack on the expansion town. If the enemy has moved lots of troops to the expansion, and has left the main base unprotected, the AI initiates an attack on the main base instead. If the AI doesn't find any suitable enemy base to attack, it queries the engine for suitable creep camps to attack instead.

```
//===========================================================
// Initiates an attack based on target priorities
//===========================================================
function LaunchAttack takes nothing returns nothing
  local unit target = null
  local boolean setAlly = true

  // Don't launch any attack while town is threatened
  if (TownThreatened()) then
    call Sleep( 2 )
    return
  endif

  // Target Priority \#1
  if (target == null) then
    set target = GetAllianceTarget()
    if (target != null) then
      set setAlly = false
    endif
  endif
  [...]
  // Target Priority \#3
  if (target == null) then
    set target = GetMegaTarget()
  endif

  // Target Priority \#4
  if (target == null) then
    set target = GetEnemyExpansion()
  endif
  [...]
  // Target Priority \#6
  if (target == null) then
    set target = GetCreepCamp( 0, 9, false )
  endif
  [...]

  // Attack the target and increment attack wave
  if (target != null) then
    call AttackTarget( target, setAlly )
    call AttackWaveUpdate( )
  else
    // If no target was found, sleep a bit before trying again
    call Sleep( 20 )
  endif
endfunction
```

Simulated computer vision could be used to improve the unit behavior. Group leaders could use it to change the formation of the group, depending

on the presence of obstacles. It could also be used to decide on attack routes and deciding where the enemy has its weak spots.

Map-making could be used to find location of strategical value, like choke points, and to reason about satiable routes for attacks and retreats. High-level path-finding could then be done in this map. See appendix A.

3.4.9 The research manager

The research manager is responsible for upgrading buildings and units. Some buildings can be ordered to *research upgrades*, such as better weapons or armor for the military units. Although it is perfectly possible to play the game without these upgrades, it is usually good to upgrade units that are used in larger numbers. The research manager needs to consult the technology tree, to know which upgrades are possible at the current time, and which upgrades that have dependencies. Since these upgrades does not introduce any new kind of problems, it will not be further discussed.

3.5 Putting it all together

When having sorted out the responsibilities of each of the managers, it is time to consider how they should cooperate. In the code shown previously, they have acted on their own, and only indirectly adapted to the actions of the other managers. To make planning possible, one should make the managers able to more directly communicate with each other, giving each other orders, requests and queries. This is unfortunately beyond the scope of this thesis. A thing to consider when deciding how the managers should cooperate is that the economic managers (all except the combat manager) goal are to provide the combat manager with troops to fight the enemy.

A listing of an AI that actually does some fighting is included in appendix C. The combat manager is very primitive though. One soldier is selected as scout, and wanders around the map. The rest of the soldiers follow the scout and attack all enemies in sight. If one put together the mechanism for creating a pathing map (see appendix A) and the mechanism for commanding troops, one has the tools to make a much more advanced combat manager.

3.6 Building custom worlds

An RTS game is normally about resource gathering and army commanding. The WarCraft World Editor can be used to create totally different maps for the WarCraft III engine though. Maps can be created that are more similar to adventure games or action games than real-time strategy games.

Chapter 4

Analysis

This project has primarily studied two worlds and how agents can be created for these using the WarCraft III engine, namely the Wumpus world and the standard, melee game world.

The Wumpus world is an example of a custom world, which took relatively short time to create. The complexity is rather low, but it is a well-known problem, described in [10]. The implemented agent is an example of a reactive agent, which creates and makes primitive use of a map to mark where it is safe to walk. The performance of the agent can be improved by introducing more rules which makes better use of the agent-created map.

Even a simple world as the Wumpus world would allow non-trivial experiments with map-making, exploration and localization, but the world would have to be a lot bigger than the shown 4x4 world. Changing the size of the world and the probability of pits in each square is luckily very easy, as the code that generates the world take the size and the probability as parameters. It would also be possible to generate the pits using other algorithms and changing the shape of the world (by adding walls and making it non-rectangular). Other changes that could be done to improve the complexity of the world is to make it evolve over time, by e.g. letting the Wumpus move. The world can also be extended to make multiple agents possible. The multiple agents can by made to cooperate with each other. This means that Wumpus world implementation in WarCraft III can be used to study exploration, map-making, planning (where is uncharted territory) and multi-agent cooperation.

The standard (melee) maps that comes bundled with WarCraft III is possible to use for studying several kinds of multi-agent problems. Even a primitive AI player as the one implemented in this study, illustrates several non-trivial problems and possible ways to solve them. The problems are both of the economic kind, gathering and spending resources to get an as good as possible army, and navigational, controlling the army in battle and scouting the map to detect strategic locations and to spy on the enemy. This means that there are non-trivial problems, including planning, map-making, exploration and multi-agent coordination, to solve.

Although WC3 can be used to study several interesting and complex problems, some limitations exists. Agents can be created both by using triggers, in the event, condition, action format, and also by using separate AI script files (both triggers and AI script files can take advantage of the relatively powerful

script language Jass). The triggers are not available in the AI script though, which makes polling necessary. It is possible to create triggers that communicate with the AI script, but the script still has to poll the communication. The alternative is to avoid using AI script and only create triggers. This is possible and usually done when creating special custom maps, but the normal way in melee games, is that the computer players are controlled by AI scripts.

Chapter 5

Conclusions

It is possible to study many kinds of AI problems using the WarCraft III engine and creating an AI that plays the standard game-type requires solving many of these problems. It is also possible to create custom worlds to be able to study even more problems. Problems that are possible to study using WarCraft III includes:

- Planning;
- Multi-agent cooperation;
- Map-making;
- Exploration;
- Path-finding.

Possible future work (which deeper focuses on parts of this project):

- Library of group-behavior, suited for multi-robot experiments.
- 1000x1000 Wumpus world. Several communicating agents (and Wumpuses?). Planning, map-making, exploration, etc.
- Symbolic reasoning about WC3 - an agent that would play better than a human, but without cheating.

Bibliography

[1] John E. Laird, "Research in Human-level AI using Computer Games", Communications of the ACM, January 2002.

[2] John E. Laird and Michael van Lent, "Human-level AI's Killer Application: Interactive Computer Games", Proc. National Conf A.I., AAAI Press, Menlo Park, Calif., 2000, pp.1171-1178.

[3] Warren S. McCulloch and Walter Pitts, "A logical calculus of ideas immanent in nervous activity", Bulletin of Mathematical Biophysics, 5:115-137, 1943.

[4] Allen Newell and J. C. Shaw, "Programming the logic theory machine", In proceedings of the 1957 Western Joint Computer Conference, pages 230-240, IRE, 1957.

[5] Allen Newell, J. C. Shaw and Herbert A. Simon, "Empirical explorations with the logic theory machine", Proceedings of the Western Joint Computer Conference, 15:218-239, 1957. Reprinted in [13].

[6] Allen Newell and Herbert Simon, "GPS, a program that simulates human thought", In Billing, H., editor, Lernende Automaten, pages 109-124, R. Oldenburg, Munich, Germany, 1961. Reprinted in [13].

[7] Herbert Gelernter, "Realization of a geometry-theorem proving machine", In proceedings of an International Conference on Information Processing, pages 273-282, Paris, UNESCO House, 1959.

[8] John McCarthy, "Programs with common sence", In proceedings of the Symposium on Mechanisation of Thought Processing, volume 1, pages 78-84, London, Her Majesty's Stationary Office, 1958.

[9] Frank Rosenblatt, "On the convergence of reinforcement procedures in simple neurons", Report VG-1196-G-4, Cornell Aeronautical Laboratory, Ithaca, New York, 1960.

[10] Stuart Russel and Peter Norvig, "Artificial Intelligence: A Modern Approach", 1995.

[11] Bob Scott, "Arhitecting an RTS AI", AI Game Programming Wisdom, 2002.

[12] "Advanced Melee AI", http://amai.wc3campaigns.com/.

[13] Edited by Edward A. Feigenbaum and Julian Feldman, "Computers and Thought", 1963.

Appendix A

Vision and map-making

Vision and map-making are problems that can be studied using the WarCraft III game-engine. There are a few limitations though. When trying to decide which objects are inside a units view, there are several kinds of objects that can't be detected. Things that can be detected include units, buildings, items and certain other things, destructables, including trees. Things that can't be detected include cliff-walls and certain structures and destructables can only be detected from map scripts, not from AI scripts. To implement vision usable for navigation therefore seems a bit troublesome if one doesn't put some restriction on how the world is created.

If one is only concerned with whether the terrain is walkable/pathable or not, there is a solution to the problem though: scanning the map by instantly moving around an invisible ghost unit. The code used by AMAI[12] is as follows.

```
function SheepScan takes nothing returns nothing
  local real x = 0
  local real y = path_bottom + path_dy / 2
  local integer i = 0
  local player neutralPlayer = Player(PLAYER_NEUTRAL_PASSIVE)
  local unit u = CreateUnit(neutralPlayer, 'odoc', 0, 0, 0)
  local real ux = GetUnitX(u)
  local real uy = GetUnitY(u)
  local real ad = path_dx/8
  local integer j = 0

  call ShowUnit(u, false)
  call UnitAddAbility(u, 'Aeye')

  loop
    exitwhen y > path_top

    set x = path_left + path_dx / 2
    loop
      exitwhen x > path_right

      set path_fieldloc[i] = Location(x,y)
      call SetUnitPositionLoc(u, path_fieldloc[i])
```

```
      if (ux > x+ad) or (ux < x-ad) or (uy > y+ad) or (uy < y-ad) then
        set path_passable[i] = false
      elseif IssuePointOrder(u, "evileye", x, y) then
        call SetUnitState(u, UNIT_STATE_MANA, 150)
        set path_passable[i] = true
      else
        set path_passable[i] = false
      endif

      set x = x + path_dx
      set i = i + 1
    endloop

    set y = y + path_dy
  endloop

  call RemoveUnit(u)
endfunction
```

Appendix B

The Wumpus world

This appendix includes code for the Wumpus world discussed in section 3.3.

B.1 A complete agent AI script

```
globals
  boolean stench = false
  boolean glitter = false
  boolean bump = false
  boolean draft = false
  boolean scream = false
  integer WALK = 1
  integer LEFT = 2
  integer RIGHT = 3
  integer SHOOT = 4
  integer GRAB = 5
  integer CLIMB = 6
  boolean exit = false

  integer x = 1
  integer y = 1
  integer dir = 1

  integer array dx
  integer array dy
  integer array aleft
  integer array aright

  boolean gold = false
  integer arrows = 1
  boolean wumpusAlive = true
endglobals

function mod2 takes integer a returns integer
  return a - ( a / 2 ) * 2
endfunction

function decode takes integer perceptSeq returns nothing
```

```
  set stench = mod2(perceptSeq) == 1
  set perceptSeq = perceptSeq / 2
  set glitter = mod2(perceptSeq) == 1
  set perceptSeq = perceptSeq / 2
  set bump = mod2(perceptSeq) == 1
  set perceptSeq = perceptSeq / 2
  set draft = mod2(perceptSeq) == 1
  set perceptSeq = perceptSeq / 2
  set scream = mod2(perceptSeq) == 1
endfunction


function print takes string mess returns nothing
  call DisplayTimedTextToPlayer(GetLocalPlayer(), 0, 0, 60, mess)
endfunction

function printPercepts takes nothing returns nothing
  local string p = "("
  if stench then
    set p = p + "stench "
  endif
  if draft then
    set p = p + "breeze "
  endif
  if glitter then
    set p = p + "glitter "
  endif
  if bump then
    set p = p + "bump "
  endif

  if scream then
    set p = p + "scream "
  endif

  set p = p + ")"

  call print(p)
endfunction

function percepts takes nothing returns nothing
  local integer cmd
  local integer data
//  call print("percepts: entering")
  loop
    exitwhen exit
//    call print("percepts: looping")
    exitwhen CommandsWaiting() > 0
    call Sleep(0.25)
  endloop
//  call print("percepts: recieving")
  set cmd = GetLastCommand()
  if cmd < 0 then
    set exit = true
```

```
  endif
  set data = GetLastData()
  call PopLastCommand()

//  call print("percepts: decode")
  call decode(data)
//  call print("percepts: exiting")
endfunction

function init takes nothing returns nothing
  set dx[0] = 1
  set dy[0] = 0
  set dx[1] = 0
  set dy[1] = 1
  set dx[2] = -1
  set dy[2] = 0
  set dx[3] = 0
  set dy[3] = -1
  set aleft[0] = 1
  set aleft[1] = 2
  set aleft[2] = 3
  set aleft[3] = 0
  set aright[0] = 3
  set aright[1] = 0
  set aright[2] = 1
  set aright[3] = 2
endfunction

function walk takes nothing returns nothing
  set x = x + dx[dir]
  set y = y + dy[dir]
endfunction

function bumped takes nothing returns nothing
  set x = x - dx[dir]
  set y = y - dy[dir]
endfunction

function right takes nothing returns nothing
  set dir = aright[dir]
endfunction

function left takes nothing returns nothing
  set dir = aleft[dir]
endfunction

function isHome takes nothing returns boolean
  return x == 1 and y == 1
endfunction

function hasGold takes nothing returns boolean
  return gold
endfunction
```

```
function climbOut takes nothing returns boolean
  return isHome() and hasGold()
endfunction

function getAction takes nothing returns integer
  local integer action = WALK

  if scream then
    set wumpusAlive = false
  endif

  if climbOut() then
    set action = CLIMB
  elseif glitter then
    set action = GRAB
    set gold = true
  elseif bump then
    call bumped()
    set action = RIGHT
    call right()
  elseif stench and arrows > 0 then
    set action = SHOOT
    set arrows = arrows - 1
  else
    set action = WALK
    call walk()
  endif
  return action
endfunction

function printAction takes integer action returns nothing
  if action == WALK then
    call print(".ai: walk")
  elseif action == LEFT then
    call print(".ai: left")
  elseif action == RIGHT then
    call print(".ai: right")
  elseif action == SHOOT then
    call print(".ai: shoot")
  elseif action == GRAB then
    call print(".ai: grab")
  elseif action == CLIMB then
    call print(".ai: climb")
  else
    call print(".ai: strange action")
  endif
endfunction

function getAgent takes nothing returns unit
  local group grp = CreateGroup()
  local unit u = null
  call GroupEnumUnitsOfType(grp, "peasant", null)
  set u = FirstOfGroup(grp)
  call DestroyGroup(grp)
```

```
  return u
endfunction

function sendAction takes integer action returns nothing
  local unit u = getAgent()
  if u == null then
    call print("Failure in sendAction")
  else
    call SetUnitUserData(u, action)
  endif
endfunction

function mainLoop takes nothing returns nothing
local integer action
  loop
    exitwhen exit
    call print("Waiting for percepts")
    call percepts()
    call printPercepts()
    set action = getAction()
    call printAction(action)
    call sendAction(action)
  endloop
endfunction

function main takes nothing returns nothing
  call Sleep(1)
  call init()
//  call decode(31)
//  call printPercepts()

  call mainLoop()
  call print("AI script shutting down")

endfunction
```

B.2 The map script

Please see `http://ai.cs.lth.se` for the code.

Appendix C

Fighting melee AI

```
globals
  unit scoutUnit = null
  real scoutX = 0
  real scoutY = 0
  real scoutHealth
endglobals

//===========================================================
//  resourceManager
//
//  Collects resources by commanding peasants
//===========================================================
function resourceManager takes nothing returns nothing
  local integer goldPeasants = 5
  local integer peasantsDone = GetUnitCountDone('hpea')
  if peasantsDone < 6 then
    set goldPeasants = peasantsDone - 1
  endif

  call ClearHarvestAI() // First reset the built in harvest-manager and
  call HarvestGold( 0, goldPeasants ) // then tell it to have
                           // goldPeasants workers harvest gold
  call HarvestWood( 0, GetUnitCountDone('hpea') - goldPeasants )
                      // and the rest to harvest wood.
endfunction
//===========================================================
//  unitManager
//
//  Trains units
//===========================================================
function unitManager takes nothing returns nothing
  if GetUnitCount('hpea') < 12 then
    call SetProduce(1, 'hpea', -1) // Train one peasant anywhere possible
  endif
  if GetUnitCount('hfoo') < 15 then
    call SetProduce(1, 'hfoo', -1) // Train one foot soldier anywhere possible
  endif
endfunction
```

```
//===========================================================
//  buildingManager
//
//  Constructs buildings
//===========================================================
function buildingManager takes nothing returns nothing
  local integer produced = 12 + 6 * GetUnitCount('hhou')
  local integer used = 1 * GetUnitCount('hpea') + 2 * GetUnitCount('hfoo')
  local integer foodSpace = produced - used
  local integer barrack = GetUnitCount('hbar')
  if foodSpace < 6 then
//    call DisplayTextToPlayer(GetLocalPlayer(),0.0,0.0,"Building house")
    call SetProduce(1,'hhou',-1)
  endif
  if barrack < 2 then
    call SetProduce(1,'hbar',-1)
  endif
endfunction
//===========================================================
//  allocateScout
//
//  Selects one unit as a scout, which will lead the troops.
//===========================================================
function allocateScout takes nothing returns nothing
  local group grp = CreateGroup()
  call GroupEnumUnitsOfType(grp, "footman", null)
  set scoutUnit = FirstOfGroup(grp)
  call RemoveGuardPosition(scoutUnit)
  set scoutX = GetUnitX(scoutUnit)
  set scoutY = GetUnitY(scoutUnit)
  set scoutHealth = GetUnitState(scoutUnit, UNIT_STATE_LIFE)
endfunction
//===========================================================
//  init
//
//  Initializes script...
//===========================================================
function init takes nothing returns nothing
endfunction
//===========================================================
//  walk
//
//  Orders the unit u to move distance forward.
//===========================================================
function walk takes unit u, real distance returns nothing
  local real dir = GetUnitFacing(u) * 3.14 / 180
  local real x = GetUnitX(u) + distance * Cos(dir)
  local real y = GetUnitY(u) + distance * Sin(dir)
  call IssuePointOrder( u, "move", x, y)
endfunction
//===========================================================
//  walk
//
//  Orders the unit u to move distance in direction.
```

```
//===========================================================
function walkDir takes unit u, real distance, real direction returns nothing
  local real dir = direction * 3.14 / 180
  local real x = GetUnitX(u) + distance * Cos(dir)
  local real y = GetUnitY(u) + distance * Sin(dir)
  call IssuePointOrder( u, "move", x, y)
endfunction
//===========================================================
//  helpScout
//
//  Sends units that follows in the scouts tracks
//===========================================================
function helpScout takes nothing returns nothing
  local group grp = CreateGroup()
  local unit u
  call GroupEnumUnitsOfType(grp, "footman", null)
  set u = FirstOfGroup(grp)
  loop
    exitwhen u == null
    if u != scoutUnit then
      call IssuePointOrder( u, "attack", scoutX, scoutY)
    endif
    call GroupRemoveUnit(grp, u)
    set u = FirstOfGroup(grp)
  endloop
  call DestroyGroup(grp)
endfunction
//===========================================================
//  scout
//
//  Controls the scout that wanders randomly on map.
//===========================================================
function scout takes nothing returns nothing
  local real R = 1024
  local real x = GetUnitX(scoutUnit)
  local real y = GetUnitY(scoutUnit)
  if scoutX == x and scoutY == y then
//    call DisplayTextToPlayer(GetLocalPlayer(),0.0,0.0,"== Bump ==")
    call walkDir(scoutUnit, R, GetUnitFacing(scoutUnit) + GetRandomReal(90, 270))
  else
    set scoutX = x
    set scoutY = y
    call walk(scoutUnit, R)
  endif
endfunction
//===========================================================
//  combatManager
//
//  Controls the army by having them follow the scout...
//===========================================================
function combatManager takes nothing returns nothing
//  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "In combatManager")
  if scoutUnit == null or not UnitAlive(scoutUnit) then
    call allocateScout()
```

```
  endif
  if GetUnitCountDone('hfoo') > 4 and scoutUnit != null then
    call scout()
    call helpScout()
  endif
endfunction
//===========================================================
//  main
//
//  Main function that runs the AI.
//===========================================================
function main takes nothing returns nothing
  call Sleep( 1.0 )  // Sleep 1 second
  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script started")
  call init()
  loop   // start eternal loop
//    call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "in loop")
    call resourceManager()
    call unitManager()
    call buildingManager()
    call combatManager()
    call Sleep( 5.0 ) //If we do not sleep sometimes, the engine will kill us.
  endloop
  call DisplayTextToPlayer(GetLocalPlayer(), 0.0, 0.0, "Script exiting")
endfunction
```

Appendix D

Command parser

This section will show the mechanisms used for implementing a command parser, useful for communicating with AI scripts and for debugging. There are two different ways to implement a command parser. It can be set up so that different triggers get called depending on the entered chat string, or the same trigger can be called for every entered string. A simple parser with few commands that take no arguments can be set up quick and easy using one trigger per command and using the graphical interface. If more advanced parsing is needed, it is easier, and more flexible, to use the textual programming and having one trigger take care of all entered text. There are ofcourse ways to combine these approaches.

To get hold of the text entered by the user, a trigger that reacts on the Player N types a chat message event. A graphical version of such a trigger could look like this:

```
Parser
    Events
        Player - Player 1 (Red) types a chat message
containing <Empty String> as A substring
    Conditions
    Actions
        Game - Display to (All players) the text: (Unknown command:  + (Entered chat string))
```

Translating this to custom text, Jass code, yields:

```
function Trig_Parser_Actions takes nothing returns nothing
    call DisplayTextToForce( GetPlayersAll(), ("Unknown command: " + GetEventPlayerChatString() ) )
endfunction

//===========================================================================

function InitTrig_Parser takes nothing returns nothing
    set gg_trg_Parser = CreateTrigger(  )
    call TriggerRegisterPlayerChatEvent(gg_trg_Parser, Player(0), "", false )
    call TriggerAddAction( gg_trg_Parser, function Trig_Parser_Actions )
endfunction
```

The entered chat string can be substringed and compared to detect commands with parameters. A command that exits the game in two different ways, depending on if a certain parameter is added to the quit command can be implemented textually like the following:

```
function Trig_Parser_Actions takes nothing returns nothing
  local string s = GetEventPlayerChatString()
  if "quit" == SubStringBJ(s, 1, 4) then
    if "victory" == SubStringBJ(s, 6, 12) then
      call CustomVictoryBJ( Player(0), true, false )
    else
      call CustomDefeatBJ( Player(0), "Defeated!" )
    endif
  else
    call DisplayTextToForce( GetPlayersAll(), "Unknown command: " + s )
  endif
endfunction
//===========================================================================
function InitTrig_Parser takes nothing returns nothing
    set gg_trg_Parser = CreateTrigger(  )
    call TriggerRegisterPlayerChatEvent( gg_trg_Parser, Player(0), "", false )
    call TriggerAddAction( gg_trg_parser, function Trig_Parser_Actions )
endfunction
```

Appendix E

Dictionary

building Constructed by workers. Treated as a unit, by the game engine.

campaign Game style which normally stretches over several maps. The game-play on the maps of a campaign can different for each map and can be more or less similar to a melee game.

map A game level or specific game world.

melee Game style where every player starts with a main building and a few workers. The goal is to eliminate all opponents by creating and using military units.

real-time strategy (RTS) game Computer strategy game where events takes place in real-time (as opposed to turn-based). An RTS game normally includes resource gathering, training of military units and warfare.

research Makes new abilities possible or upgrades special statistics of units or buildings.

RTS game see real-time strategy game.

unit Human or computer controlled entity. There exist military units, workers, and transports. The game engine also treats buildings as units.