Design and Development of MASS EXPAND, a Bot for StarCraft®: Brood War®

Ben Haanstra 5964911
Armon Toubman XXXXXX

June 22, 2012

Abstract

abstractje

Contents

1 Introduction **1**
1.1 The Game StarCraft 1
1.2 Goal of the Bot 2
1.3 Outline 2

2 Brood War AI development **4**
2.1 BWAPI 4
2.2 BWSAL 5
2.3 BWTA 5

3 Strategy **6**
3.1 General approach and AI Techniques 7
3.2 Race Selection and Sources 10
3.3 Building, Upgrading and Training 11
3.4 Macro 13
3.5 Micro 13

4 Bot Architecture **14**
4.1 Overall Structure 14
4.2 HighCommand 15
4.3 EigenUnitDataManager 16
4.4 EigenUnitGroupManager 16
4.5 EnemyUnitDataManager 17
4.6 ResourceManager 17
4.7 TaskManager 18
4.8 TaskAssigner 18
4.9 ProductionManager 19
4.10 ConstructionManager 19
4.11 ContractManager 19
4.12 MicroManager 20

5 Project Timeline **22**

6 Conclusion **23**

Chapter 1

Introduction

In this report, we are designing and developing a bot that plays the game StarCraft. A StarCraft bot is a computer program or script that plays a game of StarCraft. Our intention for the bot is that it plays without any cheats, adapts to the state of the game and responds quickly. Although we initially pursued the goal of winning as well, we decided to change the focus on being capable of playing properly.

1.1 The Game StarCraft

StarCraft is a real-time strategy game developed by Blizzard and was shipped in 1998. Its foremost expansion Brood War was released later that same year. In general, when one talks about StarCraft, the expansion is implicitly included. Over the years, the game became quite popular as it introduced various interesting gameplay and multiplayer features that easily allowed for communities to grow.

There were three totally different races to be chosen from; the technologic advanced alien race Protoss with access to psionic weaponry and energy shields, the organic alien race Zerg that thrives on evolution and its great numbers to obliterate enemies, and the humanoid race Terran that focuses on conventional weaponry such as machine guns, tanks and nuclear weapons. Each of these races have their own unique units, structures, and how they are to be played. In order to build and train those units, the resources Minerals and Vespene Gas have to be harvested and brought back to the main structure of the player.

The other foremost reason is that it was relatively easy to play online against others. This resulted in communities growing and many contests held, were money was at stake. Eventually, professional StarCraft player became a carreer for some. And in South Korea, the idea of showing games live on television was pioneered, players became celebrities, schools and training houses for teams were organized, and even an official club regulated it as a *sportsmanship*. With the fairly recent introduction of StarCraft II, however, the original game has decreased in popularity. Despite this, the game is still seen as one of the best games of all times.

The game also featured an artificial intelligence, allowing players to try a *skirmish* against the computer. The AI could play at four different levels, ranging from easy to insane. But the AI itself was a very easily programmed script. It did not adapt to the players decisions, was quite repetitive and was easily misled. Furthermore, it cheated by having full observability over the whole field, while the game itself features a *Fog of War* (areas out of sight range are blacked out). It also cheated by having additional resources, allowing it to train units even when economically impaired.

1.2 Goal of the Bot

Our goal for the bot is to develop an AI that does not cheat and actually plays the game as it should. To make it more interesting than the standard AI already present in the game, we wanted something that could actually adapt to the game and also made its own strategies.

We arrived at this idea for a project when we stumbled upon an AI competition for StarCraft, AIIDE2010, meant for bots to play each other. There were four different tournaments one could participate in, from very small scale battle to the complete game. The first two focused on team versus team battles. The third one is a simplified game of StarCraft, where both bots play Protoss, have only access to the units of the lowest tier of technology, and have to gather resources to produce units. The focus of this game is on a more strategic level, as units can not run quickly to the other side of the map. As for the fourth, we have a normal complete game of StarCraft. Although people were free to pick any race, it was adviced to only account for one specific race. The foremost reason for this is that the game is simply immensely detailed.

Initially, we started out with the idea to win the competition, but quickly found out that there is a lot more to it than writing a decent bot. Many teams of various universities signed up, some of them having more than 10 members. StarCraft is needless to say one of the more complicated games out there, and developing a very good bot involves a lot of knowledge of the game. Most of this knowledge is unfortunately unwritten and has to be accumulated by playing many number of times. Knowledge on the top level, however, is prevalently available. For instance, what structures one should build to *counter* the opponent's strategy, or what type of units are good against other type of units. In chapter 3, we go into more detail on how we gathered knowledge.

As a result, we do not aspire to claim our bot is the best, but at least hope to make a versatile bot that can adapt and also initiate combat, without cheating. Furthermore, we wanted to participate in the AIIDE2010 competition's full game tournament, to see how well it works against other bots. As we neared completion of our bot, we decided to name it *Mass Expand* referring to its tendency to create many additional basis

1.3 Outline

The outline of this project report, we first mention in chapter 2 the code made available and details of the competition. In chapter 3, we describe our bot's structure abstractally; AI techniques considered, how units should behave, what overall decisions it has to make, and how we intent to achieve an autonomously adapative bot. The chapter 4 then goes into detail on how we implement our bot's strategy. Chapter 5 exhibits our timeline of the project and interesting events. We conclude this chapter in 6, where we also reflect on our choices.

Chapter 2

Brood War AI development

StarCraft's internal AI is script-based and not very sophisticated. Since its release, the modding community has been looking for ways to create and use their own AI. One example of such an attempt is the Brood War AI Project[1], which enabled people to create their own scripts and inject them into StarCraft. The scripts allowed macromanagement (building units, general attack orders and managing resources) but not micromanagement (giving orders to individual units). The BWAPI family of libraries solved this problem by giving developers complete access to StarCraft.

2.1 BWAPI

Early 2009, a team of StarCraft enthusiasts released BWAPI (Brood War Application Programming Interface). Unlike the tools that used AI scripts, BWAPI provided programmers with complete access to the game's internals. This was achieved by reverse engineering StarCraft. While this constitutes as a third party hack to the game, and therefore violating the End User License Agreement that comes with StarCraft, Blizzard seems to condone its usage.

BWAPI is a library of classes representing game objects and methods for interacting with them. One example of the classes included is Unit, which represents units and buildings. This class has methods for getting information (like the number of hitpoints) and giving orders, such as attack orders for units capable of attacking, and build or research orders for buildings. Of course, not all buildings and units are the same. Information about what units and buildings can and cannot do and basic information that is the same for units of all types is kept in a class called UnitType. There is also a class for players which gives information about resources (among other things) and a class for that represents the game in general, with methods that for example allow bots to concede and chat with the opponent. More abstract is the Position class, used as a unified notation for positions on the game map, which can be used to coordinate the movement of units and the placement of buildings.

[1]See `http://www.entropyzero.org/BroodwarAI.html`

2.2 BWSAL

The classes and methods in BWAPI are very low level: they are small building blocks with which bots can be constructed. To get developers underway with the creation of new bots, the team behind BWAPI created BWSAL (Brood War Standard Abstraction Layer). BWSAL is an extension of BWAPI, providing classes with more high level functions. BWSAL consists of classes called managers. Each manager is responsible for a part of a bot's operation. For example, BWSAL has a Build Order Manager, which as the name implies, manages build orders. The managers can bid for control of units, which is assigned by a class called the Arbitrator. As we will later show, the idea of using managers like this appealed to us but we found BWSAL too constricting in its methodology to use for our own bot. However, BWSAL also provides two general classes called UnitGroup and BuildingPlacer, which turned out very useful and were incorporated in MASS EXPAND.

2.3 BWTA

Besides BWAPI and BWSAL, the team also created BWTA (Brood War Terrain Analyzer). BWTA is an add-on for BWAPI which analyzes StarCraft maps. It can identify base locations, chokepoints (narrow passages which can be of strategic interest) and divide maps into regions. BWTA also provides general functions such as functions to get the distance between to points on the map and functions to determine whether it is even possible to walk between to points on a map (a useful function for maps with islands). BWTA is by default included with BWAPI.

Chapter 3

Strategy

In this chapter we discuss our decisions for the strategy our bot follows. The strategy of the bot is the most important part of the bot, as not only does it involve the overall structure of the bot, but also how it makes decisions.

The *output* of the strategy, the decisions and how it plays, is in essence a *behaviour*. Based on the type of decisions, the behaviour of the bot changes. As for the *input* that induces the decisions, we view two different possibilities: perception and internal. The former is information observed or input by the player or the game, whereas the latter comes from internal variables or the mood of the bot.

We call decisions based purely on perception *reactive*, and decisions only based on internal variables *proactive*. For example, when a bot makes a decision based on the observation of five enemy units, it makes a reactive decision. And if the bot makes a decision when 'two minutes have passed', it makes a proactive one. It is also possible to make decisions based on both types of input, for example the decision to make additional units when enemy units are observed and when the bot has a mood to 'defensive'.

Each decision made by the bot (and player) has an impact on the game. When a bot sends units to the enemy, the game changes, units may get damaged and lost. The player that has still units left over, while the other does not, wins the game in StarCraft. But, it is rarely the case that all of one's units are lost in a single fight. Understanding the impact of small fights is not an easy thing to do. Even if we lose every unit in some fight, we may still have succeeded in doing enough damage that eventually would lead to winning the game.

For bots, matters are even more complicated as winning may not always be its objective. Although humans usually seek to win, bots may be designed to entertain the player. The focus is then to play interesting moves and make fights interesting for the human player.

Naturally, understanding the game and making correct decisions to accomplish an objective are the most difficult concepts intrinsic to the game. In a strategy game like StarCraft, there are many factors to be considered, the units that fight one another, the armor rate and armor type of a unit, the temporary abilities affecting the units, commands being issued, and so on.

The same holds for most other strategy games. As a result, most developers of such games do not consider to design an elaborated AI for the bot. Rather, they often follow a simple script that they execute regardless of the opponents' moves. This makes the bot quite predictive in its decisions, making it easier for a human player to find an appropriate strategy against it. And as soon as this has been accomplish, there is little 'replay' value.

Often, these bots' scripts are not very adaptive, having a simple built-in script that fully autnomously runs on a set of timers or preset conditions. Even the 'hard AI' that are sometimes implemented in games follow a similar setup. The only additional hardness is that they tend to cheat more often; extra resources, free units, little to no time required to build units, full observability (when there is a fog of war)

For strategy games, such as StarCraft, we find that bots tend to just follow a simple script. As a result, the bot plays very similarly every game, regardless of the opponent. This makes the bot predictive, making it possible for the player to easily find a winning strategy. Another issue is that these bots tend to have little to no reactive behaviour. Thus, when a player pursuits a certain strategy, the bot does not dynamically adapt at all. Arguably, this does not create much extra value to the game.

Despite our participation in the AI2010, where the focus is to win matches, we did not focus on creating a winning bot. There are two reasons for this. First, making a bot requires a lot of knowledge about the game, that we do not have. We do describe some ideas on how this could be achieved later in this chapter, but argue that pursuing the ideas require not only a lot of time but also new insights in the respective research fields. The second reason is that we would like a bot that creates interesting gameplay that is not completely predictive and adapts to the player. And it has to do without any cheating. This means that we essentially have to develop a bot that enacts as a human player.

Outline

For the remainder of this chapter, we first sketch some general appoaches and techniques that are possible to make bots play, in section 3.1. After deciding our general approach for the bot, we go into more detail on: how to build, train and upgrade units in section 3.3; How we organize units on a strategic level to attack and defend in section 3.4; and how units to control units individually and choose appropriate decisions in section 3.5. We do not describe the code itself, but rather the abstract choices relevant for the strategy of the bot. In the next chapter we go into more detail on the implementation of our bot.

3.1 General approach and AI Techniques

Research in the field of Artificial Intelligence have shown that there are various ways for a computer to play a game. We will name a few and discuss how they work. The choice of technique is important as it guides how we approach the problem of developing a bot, and ulteriorly what information and knowledge we need.

The first technique considered is related to 'board simulation' and 'board evaluation'. The idea is to first start with the current board and then, by considering various possible decisions, simulate new hypothetical boards. These hypothetical boards can be viewed as a possible future on the current board when certain decisions are made. We can repeatedly do this on the newly simulated boards, to simulate boards 'deeper'. The depth of this simulation is commonly referred to as *ply*, e.g. one decision deep is a 1 ply simulated board, and three decisions is 3 ply.

After a number of boards are simulated, we can *evaluate* them on how 'beneficial' they are for the bot. Then, by tracing back what decisions lead to the most beneficial board, we can guide our decision process. This tracing back mechanism can be viewed as searching through a Decision Tree.

Unfortunately, this technique assumes three important things. First, it assumes that we can accurately simulate boards. This requires us to know how the board will change after a decision. The second assumption is that we have a function that can evaluate precisely how beneficial the board is. And the third is an assumption that the boards themselves can be computed in reasonable time.

For our knowledge of StarCraft, we can not guarantee the first two. For the first, this would require us to fully understand the game engine and how units interact with one another. This would not only require an immense amount of time, but also demands to 'recreate' the game itself in addition to a more interesting bot. The second one, requires us to have a reasonable understanding of the game, which we do not claim to have. In fact, this is precisely what professional players of StarCraft try to achieve over many years. As last, even if we do have some idea of what is beneficial, we will have to quantify somehow to allow for a clear selection of the 'best board'.

The third assumption is more one of a computational one, which depends on our choice of algorithm and how boards are simulated. Although there are many choices, we believe that without good knowledge of the game, it is hard to avoid simulating many boards. In theory, it is perfectly feasible to simulate boards (as the game engine renders the boards themselves, too), doing this for many is computationally expensive. For example, an average game of StarCraft tends to have between 30 to 150 units, and on average a unit has 6 actions.[1]

Let us assume that at average games have an average of 80 units during the game. Then, if we consider that each unit can get issued a command per timestep of the game, we have $6^{80} \approx 1.8.10^{62}$ hypothetical boards at 1 ply deep. Clearly, this is a very large number for only a very low ply and indicates that it is clearly computationally heavy.

An alternative technique is to *learn* which decisions are good. This approach has some similarities to the previous technique. As the learning system interacts with the board, it makes random decisions and tries to find a pattern in the decisions it made and the consequent of these decisions, such as a win or loss. From the pattern found, it can conclude what decisions are good in which boards. Thus, the similarity is that it also

[1]This number of average actions per unit does not take into account that some actions may also be pointed at any unit or area in a range of the unit, such as spells of caster units.

involves 'simulation' and 'evaluation' of a board. The foremost difference though is that this approach tries to learn a *function* that chooses good decisions based on experience beforehand, whereas the previous approach simulates hypothetical future boards while playing and chooses a decision.

The problem with this technique is that it requires a lot of time. It requires to see each board several times to learn what decisions are good. And there are many different boards possible in a game of StarCraft. Even if we only consider the same number of average units before, 80, they can all be placed in numerous ways, each representing a totally different board. Even if we assume that we can come across every board, we still need to find a learning algorithm powerful enough to find a pattern in all the information. Considering the possible boards, dependencies on the past, unknown information of the opponent (as there is partial observability) and actions available, we do not know of any algorithm that can account for all this. And the development of such an algorithm requires a lot of more knowledge on how systems learn such complex problems.

Another technique that can be considered is an algorithm that learns from a human player *showing* how to play. As the human player plays the game and makes decisions, the algorithm assumes that the player makes good decisions and tries to memorize them. This is an interesting approach, but it assumes that we know someone who has time and can play reasonably well. The alternative is to use *replays* of games. They are prevalently available, but still leaves us with one other problem. Even if we do memorize what decisions are good, the bot that we develop may often find itself in a totally new situation for which it does not know what to do. Finally, it is not always easy to see whether decisions were actually good.

But this gives us some ideas on what we could do. There are several websites and forums dedicated to strategy discussion for StarCraft, from low-scale tactics to overall orders in which buildings should be built. We found most of them are quite high level, but do describe in a lot of detail on how units should move and attack. Then, if each of the units does exactly what it should do, the expectation is that the bot will play well.

This insight gave us the idea to adopt an approach of *self-organisation*, where units interact as if there are single-minded. Simply put, we expect emergence to occur from all the simple interactions of individual units. This gives two interesting benefits: first we do not need to consider to compute a whole plan for every unit, but instead can focus on how an unit by itself should work; and secondly because there is no overall plan, and units adopt by themselves, we allow for adaptivity to the opponent without writing complete explicit scripts on what to do.

Therefore, we decided to adopt this approach of controlling units individually by writing behaviour for each type of unit separately. The only necessity then is a description of *appropriate behaviour*, which is luckily prevalent on internet websites and forums.

StarCraft, however, brings some additional challenges to coding it in an self-organising way. Namely, the resources available is attached to a person and not a unit by itself. To prevent scenarios from occuring where multiple units decide 'it is good' to build new units and use resources, we must find some solutions that do not necessarily stick to the self-organisation approach.

Another point of interest is how to make sure multiple units decide to attack. Consider for instance the case when only a few units decide to attack when the enemy forces are substantially larger, or when a unit decides to attack an enemy unit when it is better when it focuses its attacks elsewhere. For this, we will consider an approach that forms groups of units, allowing for easy control on a strategic level.

In the following section we describe our choice of the race and sources. In the next sections after that we describe in detail how we decided what is good behaviour, how units are build, and how units work together.

3.2 Race Selection and Sources

In the game of StarCraft, there are three races one can select from: terran, representing the human, using technology and machinery; protoss, an alien humanoid race focusing on energy forces and psychic powers; and the Zerg, an alien race thriving on evolution to get stronger biotics that slash enemies.

For our bot, race selection is important. First, we did not expect have time to create a bot that can play all three races. Each race plays very differently and accounting for that would require an immense amount of time. Second, races require different mechanics and types of AI. Some require a lot of planning on how structures are placed, or how units should fight along each other. We decided to choose the race that is easiest to control from an *emergent* AI perspective.

The Terran focus on using ordinary guns, lasers and cannons. Most of their units are ranged and the price per unit is relatively balanced as the units compared to the other races falls in between in power. The Protoss on the other hand focus more on individual units that are very strong but also tend to be more expensive. As a result, protoss tend to have fewer units than both Zerg and Terran. Protoss units also have more abilities in general. Zerg, however, have mostly fairly bland units with little abilities. Their power comes from their numbers, as Zerg units are very cheap, but also somewhat weak. It is out of the scope of this report to give the complete details of every unit, structure and upgrade.

The three races' units obviously have to be controlled differently, and strategies differ a lot between them. But, the way the three races build structures is even more different. For the Protoss they simply *warp* buildings from outer space to the battlefield, requiring only a simple worker unit (the probe for Protoss) to put a marker for the building. The Terran, require a worker unit (an SCV) to invest time on building the structure. Naturally, if the worker is interrupted or has been killed, the structure does not complete by itself. And finally, the Zerg have their workers *evolve* into the desired buildings. This cost the Zerg a worker unit, but the interesting part is that the Zerg does not need to build many buildings.

The structures of both Protoss and Terran need to invest time in actually training units. As only one unit per training facility can be trained, players often make many more structures of the same type. This allows more units to be trained at the same time, giving the player the possibility to adapt quickly to new information of the opponent and train a new army more quickly. The Zerg on the other hand, does not have such limitations. Instead, at a short interval a larva is spawned near the main structure of the Zerg; a hatchery (and its evolved forms: lair and hive). These larva can then be commanded to evolve into a specific unit, such as a worker or combat unit. The structures of the Zerg themselves only allow larva to evolve to new types of units. As a result, each structure is only required once. To allow for more unit production, Zerg players create more hatcheries, which can also be used to gather resources.

Initially, we believed that Terran is the easiest for controlling its units. Because all of their units can shoot from a distance, we only require to move them in the *right direction*, not even having to order them to shoot at enemy units as they do so automatically. But, the management of structures is quite complicated. Not only what structure and how many, but also where to place them. We found that there is a whole science behind how buildings should be placed, as they can be of assistance in fights. For instance, preventing enemy units to access the base. For Protoss, the same holds for the structures, but the combat units themselves have to be moved more delicately, too.

As for the Zerg, however, structure placement is not important. In fact, structures are often placed such that there is a lot of space for units to move through. This is because the Zerg has often a lot of units. And the training of units is very easy for the Zerg, and the larva available is generally not a problem. Furthermore, the zerg units themselves are also easily steered. And as last, the Zerg seemed to fit the idea of *emergence* well. Therefore, we decided to create a bot that plays Zerg.

Once we have decided on the race, we tried to get our hands on good strategies against each kind of opponent. In other words, even though we selected one race that the bot would play, we still have to develop three different AIs, for each possible race the opponent may select. This is because the playstyle of the Zerg substantially differs per race.

For the AI of the Zerg, we used three different types of sources: websites on strategy discussion, feedback and knowledge elicitation from people familiar with the game, and *replay videos* of matches between professional players. From this, we could understand the game, how units can be controlled, various strategies and so on.

3.3 Building, Upgrading and Training

We will first discuss how our agent will spend resources; by building structures, upgrading technology that apply to specific units, or training new units. Regardless of the race, resources are bound to a player. That means that if resources are collected on one side of the map, we can use the resources instantly at a different location. This is an unchangeable mechanic of the game itself.

In StarCraft, there are two collected resources, Minerals and Vespene Gas, and one generated resource, supply cap. Minerals and Vespene Gas have to be gathered from respectively a mineral crystal and a vespene geyser. They are used to actually build, train or upgrade units. The supply cap effectively limits the number of units a player may have. By creating additional racial-dependent units we can increase this number. For the Zerg, this is the Overlord unit. A player can only train more units when the number of units it currently has not exceeded the supply cap.

For our bot that plays the Zerg race, we have the benefit that almost all structures only have to be built once. This means that we eventually do not have to consider building anymore and the bot only has to concern itself with what units to create.

From the knowledge we gathered from our sources, we find that there are many different strategies: dependent on the race of the opponent, the units the opponent makes, the type of map and what stage the game is currently in. In general, three stages are identified by the StarCraft community:

1. Early Game, this stage refers to the first few minutes of the game. The technology is limited and there are not many resources harvested yet. Generally, games that end during this stage either indicate a clear difference in skill between players, or when one of a player makes a hasty all-in strategy which is generally unsafe.

2. Mid Game, this stage is after the early game. Strategies for the midgame are quite varying and could either focus on getting more resources (called *economy*), having a large army (with the intention of finishing the game), or by focusing technology (so that stronger units are accessible). This stage is usually the longest as it only ends when the strongest units are brought to the battlefield.

3. Late Game, this stage starts as soon as a player gets access to the highest tier of units and technology. At the beginning of this stage, players have often a lot of resources and units available. Battles therefore become immensely large and intense. Eventually, resources may become depleted and the match turns into how well players conserve their units while destroying the other.

The community also developed a convention to communicate strategies easily: they are called *build orders*, and consist of a list of structures (and units) with a number. For example, the small build order, "10 hatch 11 pool", means build a hatchery when you have 10 supply and build a spawning pool when 11 suuply

Although a player can pursue many different strategies, we notice that the good strategies have some similarities, and may 'evolve' into other strategies. For instance, a good thing to do for Zerg is to make additional drones at the start (the worker units that can build structures and collect resources), then create an overlord (that allows for more units to be trained) and a spawning pool (gives access to the basic combat units and more technology). From this, many different strategies can be pursued, such as choosing to expand (build an additional base), create a 'hydralisk den' (to create ranged combat units) or evolve the hatchery inot a 'lair' to get access to new technology and units.

For the Zerg, the main strategy is to expand quickly and gather a lot of resources. This allows the Zerg to create more units, and use them to win by sheer numbers.

Before, we described that we should thrive on the power of emergence. This means that we essentially have the units make decisions in a decentralized manner. Each unit simply does what it can do best at the moment. But, this approach may not be so easy in the case of spending resources. If each unit has the decision to spend resources or not, we may have the problem that all resources are spent. Furthermore, there may not be a general strategy to address the enemy. Although we could solve this problem by introducing communication and negotiation protocols, we believe it may be better to have decisions related to resources made by a centralized process.

There are still some possible approaches to this centralized process. For example, we could consider to program a complete plan, e.g. a *build order* (which is like a list), that the bot simply follows as if it is a script. The problem is that if it only sticks to a certain script, it will not be very adaptive and very predictable.

A better idea is to *chop* the list into several steps, or in other words *rules*. An example rule is: "if we have a spawning pool and the enemy does not have a large army and we just started playing, then build an additional hatchery provided we have enough resources". This setup allows for easier implementation of strategies and allows for better adaptation.

3.4 Macro

3.5 Micro

Chapter 4

Bot Architecture

This chapter will describe the overall structure of MASS EXPAND and provide descriptions of the major components.

4.1 Overall Structure

The starting point of the creation of the bot was the example AI module distributed with BWAPI. The example contains the main update loop of the bot, all event listeners BWAPI supports and methods for drawing extra information on the screen. It also provides some sample code for giving orders to units. We left the example module mostly intact, except for the units orders which we removed completely. We added calls to methods of a new class, HighCommand, which would be the control center for our code. The main benefits of this were that we kept the example code for reference, and that if there would be a major change in BWAPI, we could use the new example module and quickly plug in our own code.

In our first attempts to create a bot, we tried using BWSAL, a collection of classes forming an abstraction layer for BWAPI. These classes provide methods for common tasks such as assigning workers to collect resources, maintaining a build queue and finding suitable building places. We quickly abandoned BWSAL after realizing it would be hard to adapt it to our ideas. What we kept from BWSAL were two helper classes, UnitGroup (an excellent class to select and divide groups of units) and BuildingPlacer (a class with methods to determine where a building should be built). We also kept the overall structure BWSAL used, including the naming scheme (the ”managers”).

The managers are all responsible for a part of the performance of the bot. For example, we have a manager keeping track of our minerals and gas, a manager storing information about enemy units and a manager overseeing construction of buildings and units. Organizing the bot this way allows us to seperate code pertaining to different areas of the bot operation, and also allows us to structure the way the information obtained from the game flows through our bot, resulting in commands sent back to the game. This flow of information is shown in Figure 4.1.

Figure 4.1 can also be seen as a parallelized version of the main loop of our bot.

However, in practice, the managers are called in serial form. When the update function of HighCommand is called (which is our main loop), it tells the managers to update in the following order:

EigenUnitDataManager Data about our own units is updated.

EigenUnitGroupManager Any new units we have are assigned to persisting unit-groups, small groups are merged and large groups are split up.

ResourceManager Lists of mineable gas and mineral deposits close to our bases are updated.

EnemyUnitDataManager Data about visible enemy units is updated, such as their position and health.

ProductionManager Plans are made for the construction of buildings, units, upgrades and technology.

ConstructionManager Plans made by ProductionManager are executed if possible.

ContractManager Build orders for buildings are delegated to drones and a suitable build location is found for them.

TaskManager Tasks are collected from the other managers.

TaskAssigner The collected tasks are assigned to units and unitgroups.

MicroManager Mobile units are given specific movement instructions, based on their tasks and enemies near their location.

HighCommand and the managers will be described in more detail in the following sections.

4.2 HighCommand

HighCommand is the control center of our bot. It does three things:

- It instantiates the managers and calls them in the right order.
- It listens for game events from BWAPI and forwards them to the right managers.
- It draws extra textual and graphical information on top of the game interface.

The specific order in which HighCommand calls the managers is listed in the previous section. The order is important because some managers require updated information from other managers to function.

BWAPI allows AI modules to draw extra information on the screen, on top of the game interface. This is a handy feature to show internal information of the bot that would otherwise be invisble. HighCommand currently calls these functions to display the location and type of tasks, the assignment of units to tasks (shown as lines from units to tasks), the build queue and current contracts (drones that are ordered to build a structure).

4.3 EigenUnitDataManager

EigenUnitDataManager records data about our own units (*eigen* means own in Dutch). BWAPI provides many methods to get data about our own units, but this data might change each frame. By recording data from the last frame, we are able to detect changes. One of such changes we monitored was a loss of hit points, which indicated a unit was being attacked (BWAPI later added a function to detect whether a unit was under attack). Another option this manager provides is to record which of our units were seen by the enemy and where and when this occured. If one of our units ever moved within visual range of an enemy unit, we can be sure the enemy knows about that unit and will act on that information. On the other side, we can use unseen units to our advantage (such as using them for a surprise attack).

4.4 EigenUnitGroupManager

This manager divides all our mobile units into persistent groups. Such groups can be manipulated as if it were a single unit: orders given to a unitgroup are passed to all the units in the group. Unitgroups are implemented in the UnitGroup class from BWSAL, which provides many helpful selector functions, such as "get all flying units in this group". The selector functions can be chained, allowing us to manage our units with very concise code.

The groups that the EigenUnitGroupManager maintains are updated whenever the number of units we have is changed. When a new unit is created, it is either assigned to an existing group or given its own new group. When one of our units dies or we otherwise lose control of it, it is removed from its group. The groups are then balanced by reassigning units between groups, according to various rules.

For example, new Overlords are always given a new group of their own. However, if there is a group of Mutalisks that does not have any Overlords, the Overlord will be moved to the Mutalisk group in the balancing step. It is handy to have a Overlord with Mutalisks (both flying units) because the Overlord can detect stealth units, which are then easily taken out by the Mutalisks.

Drones are a special case. We keep all Drones in a single group. While they are capable of combat, they are mostly used for gathering resources and construction. By keeping them in a special group we make sure they are not accidentally assigned to combat tasks. The drone group has a constant pointer, allowing easy access to all drones.

4.5 EnemyUnitDataManager

EnemyUnitDataManager keeps track of enemy units. By default, BWAPI only allows access to visible units. However, we we want to retain information about them when they become inaccessible, such as when they use a stealth ability or move into fog of war. Units can also transform into a different unittype. In that case, we need to make sure we recognize it as the same unit instead of counting it as two seperate units. The manager currently records five pieces of information about each enemy unit we encounter: its type, its hitpoints, its position, its last known position, and the time at which it was last seen by us. This information is saved together with a pointer to the unit.

There is an important difference between the position and the last known position. While an enemy unit is visible, we update its position, and its last known position is changed to that position. When it moves somewhere we cannot see it (for example, into fog of war) we change its position to "unknown" and keep its last known position. We can then send scouts to make that position visible. If the unit is still there, we can update its position again.

The information kept by this manager is used to create combat and scout tasks, but could also for example be used to determine the technological progress of the enemy and make predictions about the tactics the enemy will use.

4.6 ResourceManager

There are two main resources in StarCraft that have to be gathered: gas and minerals. Without these, you cannot build buildings and units, purchase upgrades and research technology. Minerals are mined from mineral patches by workers. Gas is gathered from extractors that have to be built atop geysers. Once gathered, the resources are dropped off by the workers at resource deposits. For the Zerg, the resource deposit is a Hatchery or one of its upgraded forms (Hive or Lair). Mineral patches and geysers are usually grouped together at certain locations on the map, called base locations. It makes sense to establish bases at these locations, to gain control of and allow easy gathering of the resources.

The ResourceManager catalogs the resources available at our bases. It keeps track of mineral patches, geysers and gas extractors near our bases. Since geysers are ungatherable without an extractor, it notifies the managers concerned with building when we have available geysers. The manager also creates tasks for our mineral patches and gas extractors, so that drones can be assigned to gather resources from them.

This manager needs to be kept updated for a number of reasons. Mineral patches and geysers can be exhausted after which they will no longer provide resources. Second, gas extractors are buildings and can be destroyed. Also, when new bases (expansions) are created, new resources on that location usually become available. The way this manager decides which resources are "ours" (by checking their distance to our bases) also prevents the bot from sending workers to resources being mined by the enemy, most likely resulting in the death of our workers.

4.7 TaskManager

Some of the managers (currently EnemyUnitDataManager and ResourceManager) create tasks based on their data. The job of the TaskManager is to collect these tasks, group them into categories and prepare them for assignment. Tasks have information associated with them such as a location and a target unit. The categories are:

ScoutTask Created for locations we want visible.

CombatTask Created for each visible enemy unit.

PrepareTask This task is for gathering armies, so they don't go attack in small groups which is ineffective.

DetectorTask When an enemy unit uses a stealth ability, we create want an Overlord to try and expose it at its last known location.

DefendTask Created for locations that we want to defend with extra military units.

GatherMineralsTask Created for minerals we want workers to gather.

GatherGasTask Created for gas we want workers to gather from extractors.

The TaskManager also creates some tasks itself. In the beginning of the game, it creates ScoutTasks for each base location. This results in Overlords flying across the map to find the enemy's starting location. Later in the game, when we have reached a certain number of units, we want all base locations visible at all times, to expose any expansions the enemy might have created.

4.8 TaskAssigner

The tasks collected by the TaskManager are taken by the TaskAssigner to assign them to unit groups. For each group, the TaskAssigner goes through all the collected tasks and compiles three lists: ideal tasks, good tasks and bad tasks. Tasks that are best fitted to the group are put in the ideal tasks list. Tasks that could be done by the group but are best left for another group are put into the good tasks list. Bad tasks are tasks that should not be done by this group, or may even be impossible to be performed by the group. The group is then assigned an ideal task. If there is none, it is assigned a good task. If there is no good task either, it is assigned a bad task.

What constitutes an ideal, good or bad task for a group is determined by a set of rules. For example, DetectorTasks are ideal tasks for Overlords and not even considered for other groups with other units because Overlords are the only units that can detect stealth units. ScoutTasks are always good tasks for Overlord. By not making Scout-Tasks ideal for Overlords, we ensure that DetectorTasks take priority during assignment. CombatTasks are bad for Overlords because they have no combat abilities.

Since TaskAssigner assigns tasks to unit groups, and EigenUnitGroupManager seperates Drones from the other units, Drones are also assigned tasks seperately. Currently they are only assigned gathering tasks. They are also capable of doing ScoutTasks and CombatTasks but that is not implemented in the current decision rules.

4.9 ProductionManager

The ProductionManager decides two things: which buildings, technology and upgrades (henceforth called tech) we should have, and what tech are going to build right now. These decisions are kept in what we call the wantlist and the buildlist.

We identify a number of stages in terms of at what tech we should have. Each of these stages is defined by a set of tech. When we have all the desired tech, we move up one stage. The tech defined by the current stage is kept in the wantlist. Simply put, we want the tech that is in the wantlist.

When the ProductionManager sees tech in the wantlist that we have not actually built yet, it copies that tech into the buildlist. The buildlist can be seen as an ingame build queue. The manager also adds units to the buildlist. The type of units we want to build is determined according to a set of rules. These rules take into account information such as at what stage we are (and thus what units we can actually build), enemy units that we have seen, and the race of the enemy. The list of rules is extensive, and can be said to form the heart of the strategy of MASS EXPAND.

4.10 ConstructionManager

The ConstructionManager takes the buildlist from the ProductionManager and tries to realize as much of the tech as possible. The only limiting factor is our resources.

If we can afford the first item in the list in terms of gas, minerals and supply, and we have the technological capability of building or researching the item, the order to purchase it is sent to the appropriate unit. In case the first item is a building, the ContractManager is told to create a new Contract (more on this later). We then check to see if we can afford the second item, taking into account the resources we just used on the first item. Once we can't afford an item, the process is stopped.

The ConstructionManager and the MicroManager are the only managers that send orders back to StarCraft. When we purchase a unit, an upgrade or a technology, the ConstructionManager looks up which unit builds or researches it with helpful methods from BWAPI and sends the right orders to that unit.

4.11 ContractManager

Orders to to build new buildings are given to Drones by specifying the type of building we want and the location where we want it. The Drone then moves to that location and morphs into the building. However, between giving the order and the Drone reaching the desired location, a lot can happen. The Drone could for example run into enemies and be killed or otherwise stopped from accomplishing its task. This is why we have come up with the ContractManager.

For each building we want built, we tell the ContractManager to create a Contract. Drones are then assigned to Contracts without a contractor. Contracted Drones are given a building location and are told to move there and build the desired building. The Contracts allow us to keep track of which desired buildings have been built. This way, when a Drone is killed en route to its destination, the ContractManager sees that the contracted Drone has died and will assign another Drone to the Contract (and perhaps a different building location). This method also ensures that building orders for specific buildings are given only once. In the past, there have been problems with multiple Drones taking the same orders, leading to an exodus of Drones to a single building spot, all of them blocking each other's building sites. The use of Contracts now prevent this behaviour.

4.12 MicroManager

The MicroManager is what makes MASS EXPAND move. The name comes from micromanagement: detailed management of units in combat. The MicroManager sends orders to move and attack to our mobile units. The specific orders are determined by decision trees.

There is a decision tree for each type of unit. At the beginning of the tree, the task of the unit is retrieved. Decisions are then made based on this task and the unit's surroundings. For example, Drones are usually assigned to gather the resource pointed to in their task, but may be forced to fight in case their base is under attack. Mutalisks are to engage the enemies pointed to by their combat tasks, but have to do so with special maneuvers. Overlords have to reach their scouting tasks, but avoid anti-air defenses. There are too many details to describe here.

The manager usually iterates through all units and runs them through the decision trees. To speed up this process, we have given the manager the ability to grab nearby groups of units and give them the same orders. This is especially handy in combat situations, because enemy units can be brought down fast with focus fire.

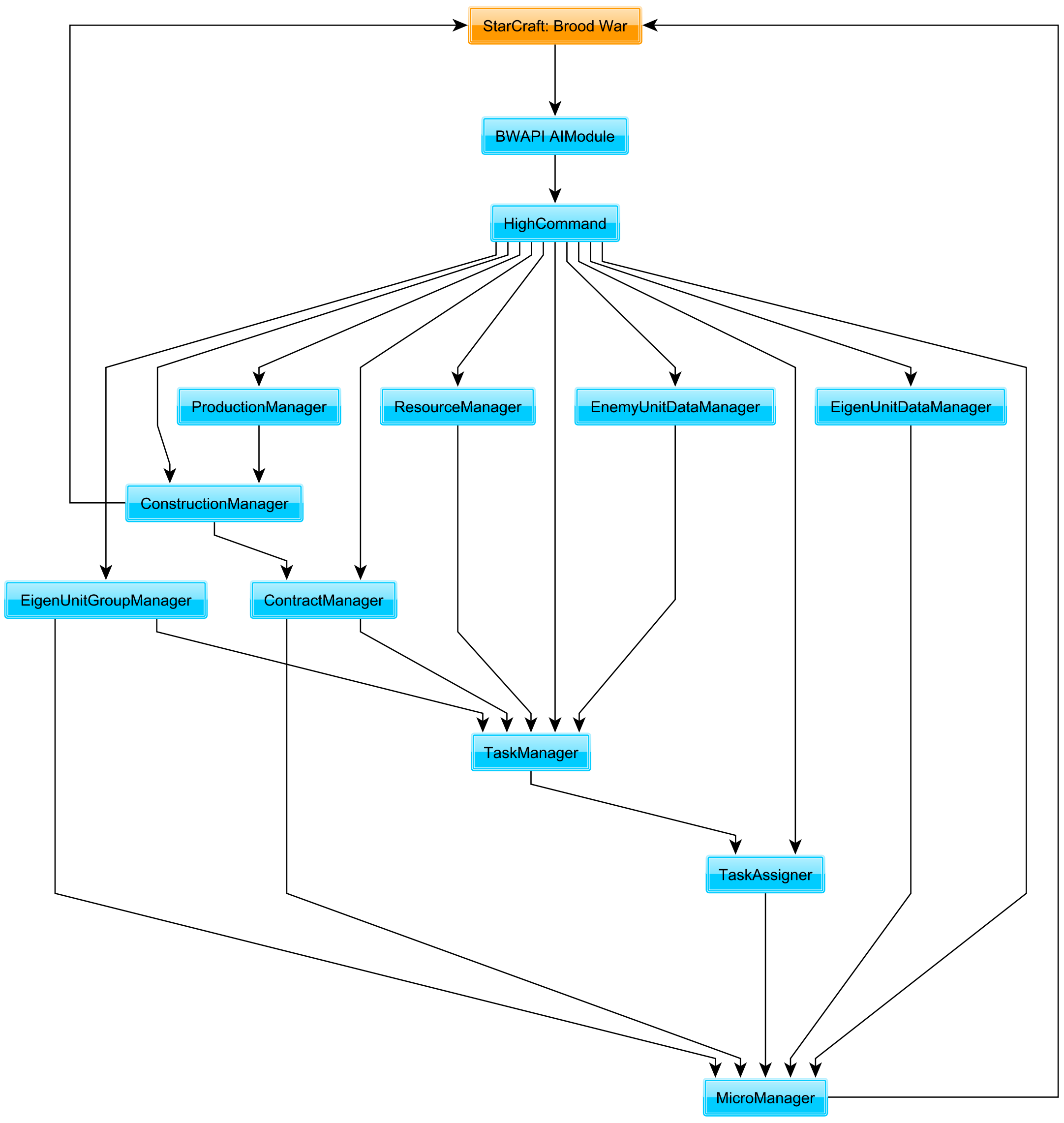

Figure 4.1: The flow of information from StarCraft through Mass Expand to MicroManager and ConstructionManager, which send commands back to StarCraft.

Chapter 5

Project Timeline

January 2010 Found an announcement for the AIIDE 2010 StarCraft Competition. Team MASS EXPAND is born.

February 2010 - June 2010 The Game Programming course is given at the University of Amsterdam and followed by both team members. A proposal to submit the bot as the final project for the course is accepted.

August 2010 - September 2010 The first version of MASS EXPAND is ready. The bot plays well but suffers from slowdowns and crashes.

September 2010 - October 2010 The competition is held at UC Santa Cruz in California, USA. MASS EXPAND participates in tournament 4 (complete games).

November 2010 The results are posted online. MASS EXPAND won the first round 1-3, but then lost 0-3 to the bot which would continue to win the tournament.

August 2011 Work continues on a second version. The code is rewritten from scratch and uses the latest version of BWAPI, hoping to eliminate the slowdowns and crashes. This version will be submitted for the Game Programming course.

September 2011 Finishing work on the bot and the documentation.

Chapter 6

Conclusion

reflection, conclusion, results

Resources

[1] MassExpand project page.
http://code.google.com/p/massexpand/

[2] MassExpand code documentation.
http://www.armontoubman.com/massexpand/html/index.html

[3] BWAPI project page.
http://code.google.com/p/bwapi/

[4] BWSAL project page.
http://code.google.com/p/bwsal/

[5] BWTA project page.
http://code.google.com/p/bwta/

[6] AIIDE 2010 StarCraft AI Competition page.
http://eis.ucsc.edu/StarCraftAICompetition

[7] Blizzard Entertainment: StarCraft.
http://us.blizzard.com/en-us/games/sc/