Real-time AI in video games: decisive and collaborative actions

Stanislas Dozias, Gaspard De Batz, Thibaut Vacek

1 Introduction

In this report we will explain how the artificial intelligence we have coded works. This includes the overall structure of the code, how the different .cpp files communicate with each other, but also the choices at a smaller scale, typically moving a unit or constructing a building, which is more of a decision. We chose the Zerg race.

2 Code's structure

2.1 General idea

The action of the autonomous agent is divided into 5 files which more or less correspond to the different scales of decision making.

- First of all, there is the "StarterBot" file, which is only used to link the files and trigger some code at relevant moment (death or completion of a Unit for example). It is called by the main to launch the decision analysis of our AI, and sends and retrieves the information needed by the other files.
- Then we have the "Scenario" file, which decides the main strategies chosen by our AI. It analyses the situation: at what stage of the game are we for example (to select the good units that we want to create). At first we had the idea of more complex analysis (what is the opponent's race, is the opponent attacking, with what type of attack should I attack, etc.), but we couldn't implement all of them.
- Then we enter a smaller scale but not yet at the level of micro-management. The "Actions" file, is the lieutenant of the general "Scenario". It is used to translate the major axis chosen by the scenario into smaller scales. If action choose which is the next unit we want to morph from larva, it is up to Actions to concretely give the order to a larva to morph to the good type. It is also this file that divides the units into "squads" (our structure which manage groups of units), and then assigns tasks to these groups of units.
- Finally, one or more .cpps take care of the micro-management ("Squad" for the management of a group of unit and "MicroGestion" for each unit). In other words, once the bot has made a decision, has distributed tasks and ways of doing things to the units, these files translate what such information means for a unit performing an action per frame (move, attack, detect enemies, ...).

The structure shown in the diagram resumes this general idea. The details of each file overlap with the explanations in this paragraph but go deeper into the final coded functions.

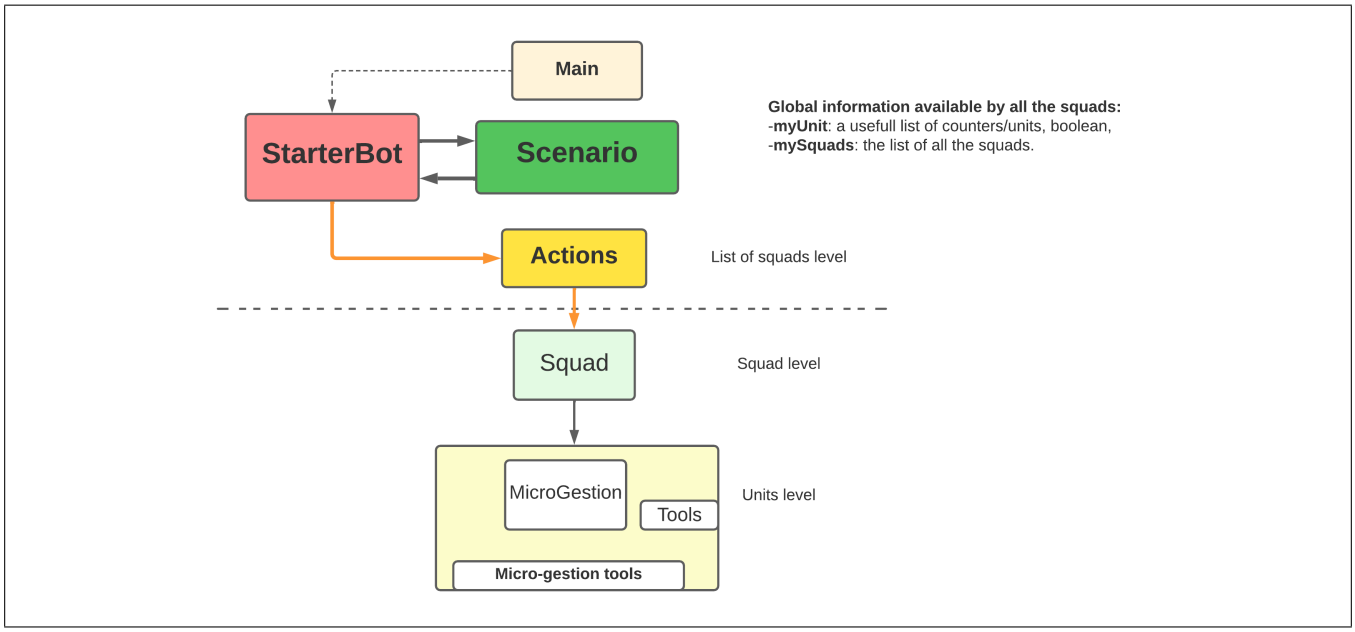

Figure 1: General structure of the code

2.2 Details of the roles of principal files

2.2.1 Global Variables

Let's first explain the two global variables:
- "myUnits", which contains a bunch of counters, units and booleans, useful everywhere in the code.
- "mySquads", which is the list of all the squads we own, allowing our functions to browse among them (this is more robust than having the squad in "hard" in myUnits). It contains a "blank" squad (the first, targeted in the code by "mySquads.front()"), which get all completed units. These units are after distributed to the good squads (via the function "enlistUnit" in the file Action). We didn't succeed to make it concretely a global variable, but it is as it is the case.

We have also imported a package (BWEM), which analyses the map. It is used to find the location of the second base.

2.2.2 StarterBot

The StarterBot file is the big coordinator. It launches the major actions of analysis, decision, and actions distributed in the other files. It coordinates all these steps by collecting the information updated by the other files, which will later be used by smaller scale actions. The main idea is to ask the scenario what the current production objective is, and then send all the information to Action.cpp so that it can adapt itself.

2.2.3 Scenario

Scenario.cpp is mainly used to set the target for unit production. In particular, it performs three actions:
- via "countUnits" it updates the object "myUnits". The update is mainly on the units owned and morphing. It also detects the failures of building and allow their catch-up.
- it decides the production objective (how many lurkers, Zerglings, etc. do we want to own).
- finally, it chooses the unit into which the next Larva will morph

2.2.4 Actions

The Actions file is separated into two types of function: the "real" actions which are called by Scenario and other functions which are mainly about the management of several squads and are useful for the actions..

Here is an overview of the different Actions:
- "morphFromLarva" morphs the current Larva into the good unit (and handle the creation of Overlord when necessary).
- "Economy" manages the squads of Drones. It affects them to minerals or to the right Extractor. It also makes Drones attack if an enemy is nearby to help the defense.
- "buildHatchery" handles the construction of hatcheries, which are of two types. First there is the one for the harvesting of the second base (which allows to have a second gas Extractor and more Drone to mineral). The other one are here to produce more Larvae and so be able to produce more units. The mineral threshold to build these "Macro-Hatcheries" increases with the number of hatcheries owned.
- "buildTechTree" carries the decisions of tech evolution. You can find below the "Tech tree". Our code has been made so that if we loose one of the building of the Tech Tree, we will rebuild it.
- "baseArmy" manages the army. It defends until it has reached the number of unit wanted for each Types. Once the attack is launched, we specify that the Army Squad is already attacking, making all the units inside permanently attack. Also, the units wanted for the squad army are set to "unlimited" (to get all the combat units produced by the Larvae).

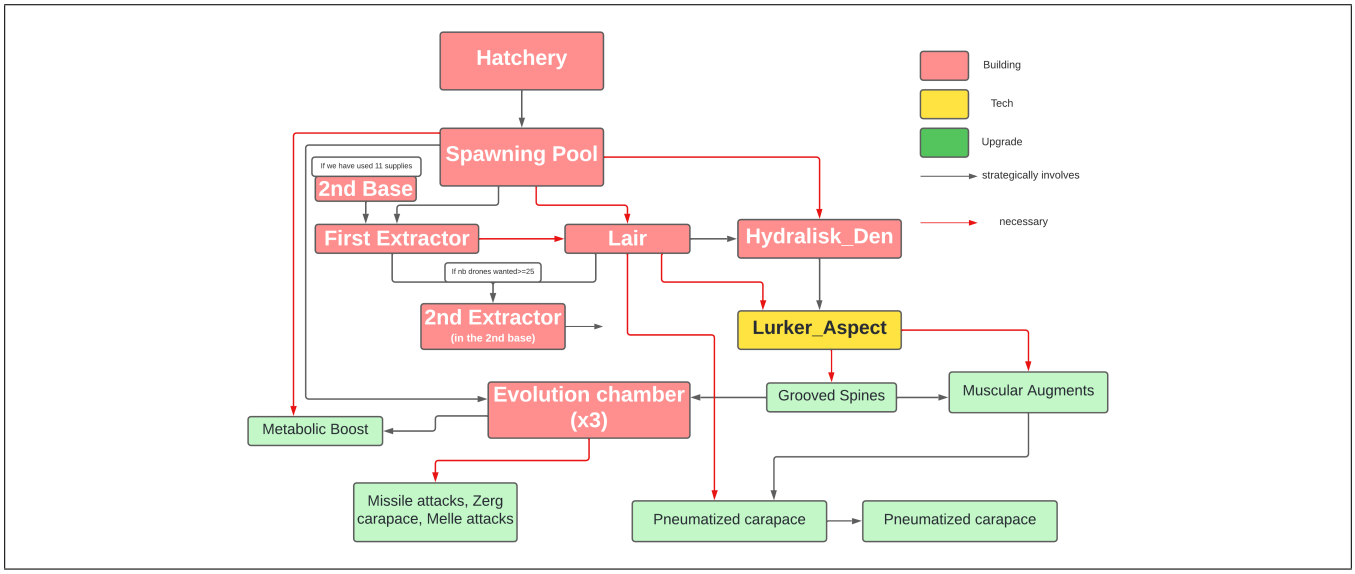

Figure 2: Tech tree

2.2.5 Squad

General presentation

Instead of the previous files which manage all the units and all the squads, the Squad file is used to manage one Squad of units and centralize their orders. A Squad include the following attributes:

- The type: either "Worker" or "Army", which correspond to harvesting or attacking squads.
- The action: useful only in the "Worker" type : the function defines the harvested resource (minerals or gas) and the base attached (only for gas). For "Army" type, we can imagine different functions, like principal army v.s. harassing army for example, but for now we only have one big Squad for all the combat units.
- The counters of unit (unit owned, unit morphing and unit wanted)
- the Army Squad also has a boolean which indicates if the squad is already attacking or not, to allows a continuous attack

Attack

The class contains a very important function: the one that manages the attack of the Army squad on the enemy. Here is how the attack is carried out:

- if a unit of the squad detects an enemy unit, all the unit attack the target position
- else, it means that we didn't have met the enemy, and we have to manage a little our units. Indeed, they do not have all the same movement speed : the Hydralisks are slower than Zerglings for example. To prevent a "double" attack with first the Zerglings alone and the Hydralisks/Lurkers after, we detect the closest Hydralisk from the target position, and all the unit which are not Hydralisk attack toward instead. You can see it in action with the image below.
- the function manages also the behavior of Lurkers (which can attack only when burrowed, but then they can't move) and Overlords (only here to detect invisible units) which are quite different. The Lurker follow the following rule : if he detects a unit closer than its range attack, it burrows (if unborrowed) and so can attack it, else it unborrows (if burrowed) and move to the target position.

2.2.6 Microgestion

The Microgestion file offers different functions at the unit scale:

- build a unit with a specific unit (or choose one in the mineral gatherers if none given).
- detect enemies close to a unit

2.2.7 Tools

This file was already implemented in StarterBot. We only added some usefull functions and changed a little some of them.

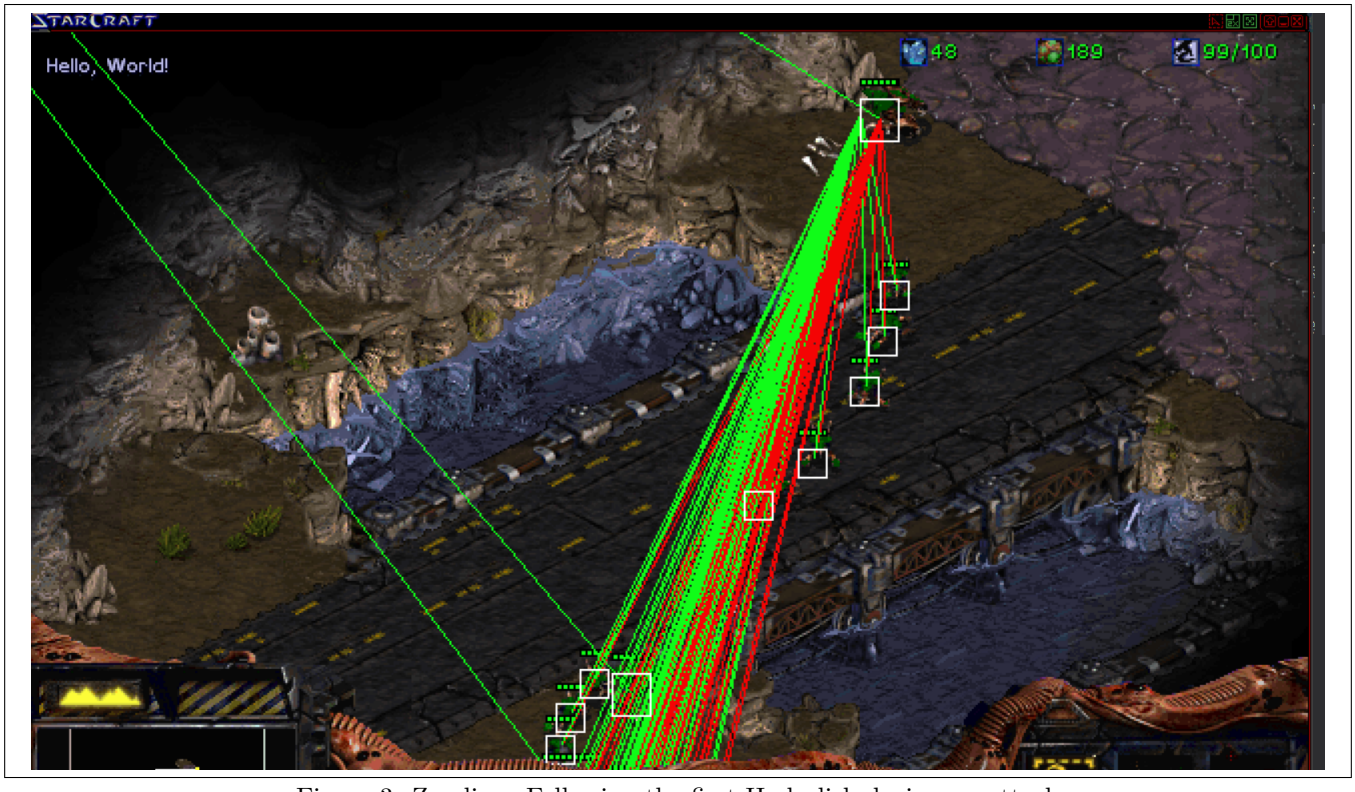

Figure 3: Zerglings Following the first Hydralisk during an attack

3 Strategy

3.1 General strategy

We have previously shown some aspects of our strategy, which concern different aspects of the game (resource management, attack, technology evolution...). It is however important to talk about our overall strategy, which follows the following action plan:

1. Taking safely and swiftly a second base, in order to have a good economy:
   - Make a Spawning Pool at 9 Drones (before morphing an overlord)
   - Continue morphing Drones Until the Spawning Pool finishes
   - Morph until 10 Zerglings while taking the second base (we have enough drones and not to much larvae to make both at the same time).

2. A "rush" to the Lurker Tech, in order to have a strong defense thanks to this powerful unit which attack in a line (and so hit several enemies) and is invisible (the opponent has to get detection to see it):
   - Take a first Extractor when the second base is launched
   - Morph Drones until having 15
   - Begin the construction of 10 Hydralisks and 4 lurkers (we research the technology needed while doing so)

3. A raise of the economy and an augmentation of the Army Wanted (because we are safe):
   - when we have 2 lurkers (we consider us safe), we set the Drone wanted to 30 to develop a lot the economy (we make them in priority to others units)
   - The Aim of Zergling is set to 50, Hydralisk to 20 and we keep an aim of 4 lurkers
   - we continue in the Tech tree with a lot of upgrades, like the powerful ones in the Evolution Chamber

4. Attack: When the number of combat units is enough, the attack is launched. The objective is to drown the opponent under the large number of attackers produced, knowing that the easy production of units is one of the advantages of the Zerg race and that, because we got a second base early, we have a large income.

3.2 The special case of the "Pool Rush" strategy

It is important to note that our bot is designed to achieve the right balance between long-term development and short-term survival. In other words, our bot aims to reach phase 3 of our general strategy, but sets up early enough in the game a defense capable of resisting a quick attack from the opponent. This concerns in particular a strategy sometimes adopted by the Zerg: this race has the advantage to be able to produce a wave of attackers almost from the beginning of the game. Our bot is designed to resist this type of strategy, which focuses on the first moments of the game. This can be seen by the safeness when we take our second base : we set up a bunch of Zerglings while doing so to be ready in case of an attack, and in the defense function, we wait to have enough (6) defending units to defend the second base. The purpose of this is, in case of a Zerglings rush, to have the Drones helping our Zerglings during the enemy's attack.

4 Data structures and algorithms coded in the AI

Our artificial intelligence is mainly made up of condition trees, because in our conception of strategy, this was the most suitable structure, and the only one really necessary. At one point we thought of the blackboard solution to synchronise the general scenario and the more micro tasks, but this concept was not necessary in the end, as the conditions were already largely effective and allowed us to go far in the complexity of the strategy.

No fuzzy logic is found in our artificial intelligence, again because it is not needed. Indeed, each action decided is wanted at the moment it is triggered and the goal is to perform it as quickly as possible. We have not included any scaling in important actions of our strategy.

On the other hand, many evaluation functions are coded, in particular to update very regularly and well foresee the resources possessed, the units possessed, wanted, in production...

Also, Hierarchical Finite State Machine code can be found in many types of management. For example, the Lurkers that alternates between hiding underground to attack and moving can be seen as a very basic HFSM, where the states are (attack/burrowed) and (move/dug up) and the transitions depend on the presence of nearby enemies.

5 Conclusion

To conclude, we have tried in this document to make you understand what our general ideas are for building a strategy that is strong enough to withstand a wide range of opposing strategies. We have shown how these ideas were translated into our code and into the final artificial intelligence, which is ultimately a giant tree of conditions. We also have tried to have the most general code, which allows it to easily adapt it if we want to add some functionalities. For example the defence of the "Zergling Rush" was possible in only two hours because of it.