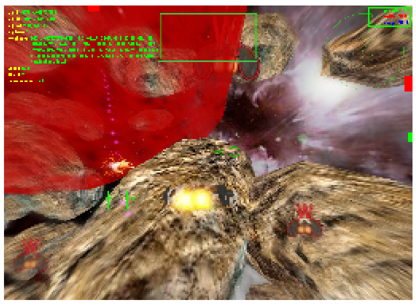
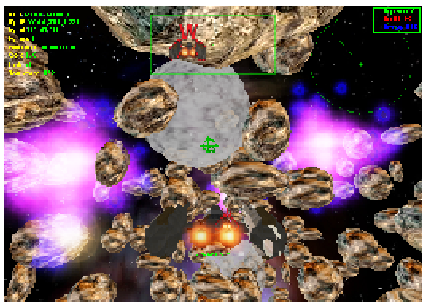
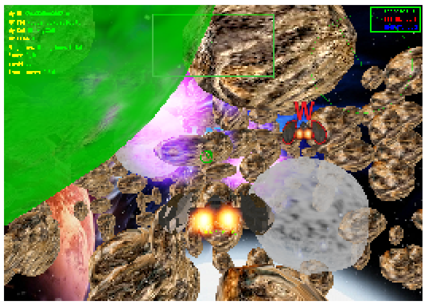
Artificial Intelligence for a Massively Multiplayer Online Game

Künstliche Intelligenz für ein Massively Multiplayer Online Game
Bachelor-Thesis von Dimitri Wulffert
März 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

DVS

Artificial Intelligence for a Massively Multiplayer Online Game
Künstliche Intelligenz für ein Massively Multiplayer Online Game

Vorgelegte Bachelor-Thesis von Dimitri Wulffert

1. Gutachten: Prof. Alejandro Buchmann
2. Gutachten: Max Lehn

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 31st March 2011

(Dimitri Wulffert)

Abstract

Applying the peer-to-peer architecture to massively multiplayer online games (MMOGs) is a concept that until now didn't reach any commercial game. The idea of creating a game that allows players to play MMOG over a peer-to-peer system is a great step forward for the future: it will allow small companies to create MMOG without having to sustain the cost of a server infrastructure or to request a monthly subscription to the players. On the other hand gamers will be more attracted to such games since they will not have to pay monthly fees to play a peer-to-peer MMOG.

Aim of this work is to create a new AI for a peer-to-peer MMOG called Planet $\pi 4$. First I analyzed the current state of art of the AI for computer games. Then I considered two different approaches to create the AI for Planet $\pi 4$. The first AI implements a simple behavior while the second AI creates a more complex one. This is done in order to test if the complex AI plays in fact better than the simple AI. This thesis includes several simulations to verify both which AI is the best and to see how well each AI adapts to the game. Creating such AI for a peer-to-peer MMOG is necessary to simulate Planet $\pi 4$ with thousands of peers. Each of these peers will be controlled by an AI and such AI should simulate human behavior. Finally this thesis makes some recommendations on how to improve the game and the current AI for future work.

Kurzfassung

Bislang gibt es noch keine kommerziellen MMOGs (Massively Multiplayer Online Games), die auf einer Peer-to-Peer-Architektur basieren. Die Idee, MMOGs über ein Peer-to-Peer-System zu spielen, stellt einen großer Schritt in die Zukunft dar: es ermöglicht es kleinen Unternehmen MMOGs zu entwicklen, ohne die Kosten für eine Server-Infrastruktur tragen oder von Spielern eine monatliche Gebühr verlangen zu müssen. Ebenfalls sind solche Spiele, für die keine monatlichen Gebühren gezahlt werden müssen, für eine größere Anzahl an Spielern interessant.

Ziel dieser Arbeit ist es, eine neue Künstliche Intelligenz (KI) für einen Forschungs-Prototyp eines Peer-to-Peer-MMOG namens Planet $\pi 4$ zu erstellen. Zunächst analysiere ich den aktuellen State-of-Art der KI bei Computerspielen. Dann entwerfe ich zwei verschiedene Ansätze, um die KI für Planet $\pi 4$ zu erstellen. Die erste KI weist ein relativ einfaches Verhalten auf, während die zweite KI sich komplexer verhält. Der Zweck dieser Vorgehensweise ist es, zu überprüfen, ob die komplexe KI besser als die einfache KI spielt. Diese Arbeit umfasst mehrere Simulationen, welche analysieren, welche KI besser funktioinert und welche sich besser an das Spiel anpasst. Die Erstellung einer KI für ein Peer-to-Peer-MMOG ist notwendig, um Planet $\pi 4$ mit Tausenden von Peers testen zu können. Jeder dieser Peers wird von einer KI gesteuert und sollte imstande sein, menschliches Verhalten zu simulieren. Schließlich gebe ich in dieser Arbeit Empfehlungen, wie man das Spiel sowie die entwickelte KI in Zukunft noch weiter verbessern kann.

Contents

1 Introduction **5**
- 1.1 Goals . . . 5
- 1.2 AI . . . 5
- 1.3 MMOG . . . 6
- 1.4 Outline . . . 7
- 1.5 Glossary . . . 7

2 Related Works **8**
- 2.1 Planetπ4 . . . 8
  - 2.1.1 Plot . . . 8
  - 2.1.2 Game-play . . . 8
  - 2.1.3 Planet π4 over peer-to-peer . . . 10
- 2.2 AI State of art . . . 10
  - 2.2.1 Rule-based systems (RBS) . . . 10
  - 2.2.2 Finite-state machines (FSM) . . . 11
  - 2.2.3 Goal-oriented action planning (GOAP) . . . 11
  - 2.2.4 Hierarchical task network (HTN) . . . 12
  - 2.2.5 Behavior Trees (BTs) . . . 12

3 Concept **14**

4 Implementation **15**
- 4.1 AI Architecture . . . 15
- 4.2 AI controls . . . 15
- 4.3 ExtBot . . . 17
- 4.4 IBot . . . 17
- 4.5 FSM AI . . . 17
  - 4.5.1 State . . . 18
  - 4.5.2 State Machine . . . 18
  - 4.5.3 Interfaces . . . 19
  - 4.5.4 Implemented states . . . 19
- 4.6 Goal Oriented AI . . . 22
  - 4.6.1 AI_Dragon_Bot . . . 22
  - 4.6.2 Goal . . . 22
  - 4.6.3 Implemented Goals . . . 24
  - 4.6.4 Primitive Goals . . . 26
  - 4.6.5 Composite Goals . . . 27
  - 4.6.6 Goal Evaluators . . . 28
  - 4.6.7 Execution . . . 31

5 Evaluation **33**
- 5.1 Evaluation Goal . . . 33
- 5.2 Metrics . . . 33
- 5.3 Simulation scenario . . . 33
- 5.4 Simulation Environment . . . 34
- 5.5 Simulation results . . . 34
  - 5.5.1 First Simulation . . . 34
  - 5.5.2 Second Simulation . . . 37

5.5.3 Third Simulation 39
5.5.4 Fourth Simulation 41
5.5.5 Fifth Simulation 43
5.5.6 Sixth Simulation 45
5.6 Summary of results 46

6 Conclusions **47**
6.1 Summary 47
6.2 Future Work 48

1 Introduction

Computer gaming is becoming more and more popular every year [19]. They offer a totally different game experience, depending on which kind of game the player is playing. Games in these days usually offer two different modes to play them, single-player and multi-player. In single-player mode, the player only interacts with the Artificial Intelligence (AI) of the game and with the given environment. On the other hand multi-player games do not only interact with the same elements of the single-player, they also interact with other players via network. Depending on how many players are playing on-line, we can talk of a massive multiplayer online game (MMOGs) when more than 1000 players are playing together

Most of the MMOGs rely on a client-server system, which in most of the cases works pretty well [19]. Problems with MMOGs usually occur when servers become overloaded and cannot handle all the incoming traffic. In this case players experience a problem called lag, in which the game instance of the players is not correctly synchronized with the server game instance. Another problem with this system is the economic part, not only for the company but also for the costumer. In order to maintain this kind of architecture functioning for massive amounts of players, companies have to escalate and update the infrastructure and hardware of the system according to the game's population growth, which means that the running cost of the service will increase. As a result, users usually are required to pay a monthly fee in order to play the game. So if we look at the client and server sides, both have to pay for the running costs of the servers and this cost get more expensive with the increase of the population of the game.

In order to improve this model and make MMOGs more reliable and affordable, not only for small game companies who do not have the economic resources to maintain the client-server architecture, but also for gamers who wish to play MMOGs without a monthly fee, the peer-to-peer paradigm comes as a serious solution to this problem.

Peer-to-peer is the most adequate candidate to solve these problems, not only because is the most economical solution, but it can also provide the same or better performance than a client-server system.

1.1 Goals

Looking at the current state of commercial MMOG, most of them are client-server based [24]. So the main goal of this project is to present a solution that only uses peer-to-peer and gives equal or better performance than a client-server system. In order to achieve this goal, the following sub goals have to be accomplished:

- The core of the game must be completed. This means that the game has a specific objective, which must be completed in order to win the game.
- The simulator simulates a peer-to-peer network, which handle more than 1000 peers.
- Each peer runs an AI, which plays smart enough to work with other AIs.

The main goal of this thesis is to create an AI smart enough to challenge human players. This also should be able to handle messages that are in the network, in order to work as a team with others AIs.

1.2 AI

Artificial Intelligence is highly related to human behaviour. Most of the research based in AI has some relation with behaviours or actions that humans usually do. For instance voice recognition, robotics, vision or creating artificial opponents/allies for computer games, etc.

The most commonly definition of AI was made by Alan Turing [3]. This definition is based on the Turing test which goal is to prove that the machine, that is currently being tested, contains intelligence. If the machine contains intelligence, it also has to demonstrate it.

The test works as follow:

Imagine that there is a person and a machine, between the two of them there is a curtain. The person starts to talk to the computer and the computer will also talk to the person. If the person does not recognize that the machine is a machine, then this machine had successfully accomplished the Turing test. This means that the machine is intelligent enough, to be confused with a human.

AI in computer games is different than the definition of the academic world. In the academic world the AI wants to show how intelligent it is, but in computer games, the AI only has to show an illusion of intelligence. This illusion could be achieved by showing the player a challenge, in order to win the game. The sense of challenge comes from how smart the AI of the game plays against the player and not only from how strong the enemies are. In games like Halo [27], the AI doesn't get smarter as the difficulty increases; only the enemies become stronger, which increases the challenge at higher difficulties and the game could be enjoyed equally in all difficulties.

Looking a few years back, most of the games were compared by their graphics. Back then each year the improvement in the graphics was great, showing better textures quality, shaders, particles effects, etc. At that time AI in games didn't have much relevance, as it is now. The public was happy if the game had some nice graphics, nice game play and somehow an ok AI. Nowadays it is difficult to see how much the graphics has been improved since last year, so the public is starting to look for other factors, when buying a new game. Here is where AI comes in play. Now most of the companies invest more time and resources to implement an outstanding AI, in order to outsell the competition.

AI for computer games depends a lot on which kind of game is implemented. For instance if the AI is implemented for a first person shooter (FPS), the AI will have in mind, which tactics are important when playing a FPS, like don't go to the battlefield under open fire or to use walls in order to protect itself from enemy fire. This changes from game to game since the structure of each game is different.

Another important aspect of the AI in computer games is that the AI should be at all times fair. This is very important since the AI has many abilities that humans don't posses. Abilities like perfect shooting or knowing the exact location of a player are not possible for a normal human. Using these strategies in order to overwhelm human players, are considered as a cheating AI, making the game less fun to play.

1.3 MMOG

Massive Multiplayer Online Game (MMOG) is a multiplayer game that runs online with massive amount of players. The genre has become very popular since the success of World of Warcraft (WOW) [9]. MMOG contains many different kinds of games, like First Person Shooter (FPS), Real Time Strategy (RTS), racing and Role Playing Game (RPG). Although at the moment, the most popular genre in MMOG is RPG, this may be due to the success of WOW and other free to play MMORPGs.

When the MMOG genre started, the number of players wasn't massive at all, in comparison to how it is now. Nowadays games like WOW handle more than 100.000 players per server which means a massive amount of information running though the network and consuming hardware resources. In order to be able to maintain a good quality of the game experience and to cover the resources and infrastructure costs, most game companies demand a monthly fee in addition to the initial cost of the game. This payment of a monthly fee is not seen positively by players.

On the other hand, several free to play MMOGs are available in the Internet but the quality of these games is not comparable to the quality of commercial games like WOW. One of the few exceptions is Guild Wars [1] which is free to play after buying the game and was classified as a great game by many reviewers [17]. The limited existence of non commercial MMOGs offering a great quality of game, makes MMOGs running over a peer-to-peer system very appealing.

1.4 Outline

The outline for this thesis is structured as follows: in Chapter 2 the related works for this thesis are presented, starting with Planet π4 the game in which is based this thesis. It also explains the technological background for AI in computer games and how peer-to-peer is related to Planet π4, MMOG and how peer-to-peer is related to Planet π4. In Chapter 3 we propose two AIs for MMOGs over peer-to-peer. In Chapter 4 we explain in detail the implementation of these AIs. The evaluation of the implemented solution based on simulation results is presented in Chapter 5. The thesis is concluded in Chapter 6 summarizing the work and presenting the next steps that should be considered for future work.

1.5 Glossary

AI - Artificial Intelligence

Gameplay - describes all possible interactions between user and game software. Gameplay does not consider graphics and sound.

AOI - area of interest. In the context of this work AOI is always the area near some subject, for example a ship's AOI is the area in which other ships are visible for it.

Bot - AI which plays a game and tries to simulate human behaviour.

HUD - head-up-display, shows the most important information to the player.

Instance - an instance of a program is its representation in the computer's memory.

MMOG - massive multiplayer online game.

P2P - abbreviation for peer-to-peer.

POI - point of interest. It can be an upgrade point or a shield regenerator.

Respawn - to respawn <-> enter the battlefield again, shortly after your ship was destroyed.

FSM - Finite State Machine

GOAP - Goal Oriented Action Planning

HTN - Hierarchical Task Network

BTs - Behaviour Trees

2 Related Works

This chapter introduces the technological background of this thesis. It starts with a description of the peer-to-peer game Planet π4, on which the work of this thesis is based. The chapter ends describing the state of the art of Artificial Intelligence (AI) for computer games including the different relevant AI systems.

2.1 Planetπ4

Planetπ4 is a research prototype MMOG. The propose of this game is to serve as a peer-to-peer Benchmarking. This allows to evaluate how well MMOG works over a peer-to-peer network.

2.1.1 Plot

The game takes place in outer space near a planet called π4. The planet is surrounded by a ring of asteroids, which contains massive amount of resources and a new form of resource called Energy Field, making it a perfect place for a mining field.

Over the last 200 years many nations around the galaxy fought to take control over the asteroid ring of planet π4. After fighting for so long, the asteroid ring was not a mining field anymore; it became a battle field.

In order to take advantage of the Energy Fields, different nations started to create assault ships, which can absorb the energy of the Energy Field to improve and repair their ships.

2.1.2 Game-play

The basic idea of the game is that the player has control over an assault space ship in a 3D world. The ship can fly in any direction and shoot enemies in sight. This is possible by using the following controls:

Keyboard:

- Accelerate (Key W): increases the current speed of the ship.
- “Boost” (Key Space): increases even more the current speed of the ship.
- Reverse thrust (Key S): allows the ship to flight in reverse or to reduce the ships current speed.
- Strafe (Key A and D): allows the ship to slide to the side (left or right).

Mouse:

- Shoot (Left button): fires the equipped weapon of the ship.
- Aim (Mouse): each ship has a crosshair associated which can be used to control either the direction of the ship or to aim enemy targets. The crosshair is controlled by the mouse.

As shown in **figure 1**, the game contains a graphical user interface (GUI), which shows the current status of the ship:

1. Upgrade: number of upgrade points that the team controls at the moment.
2. Health: amount of health of the ship. This amount can vary between 100% and 0%. A decrease in health can be reached, either by getting hit by energy bullets or by colliding against other objects (ships or asteroids). When 0% is reached, the ship will show a nice explosion animation.

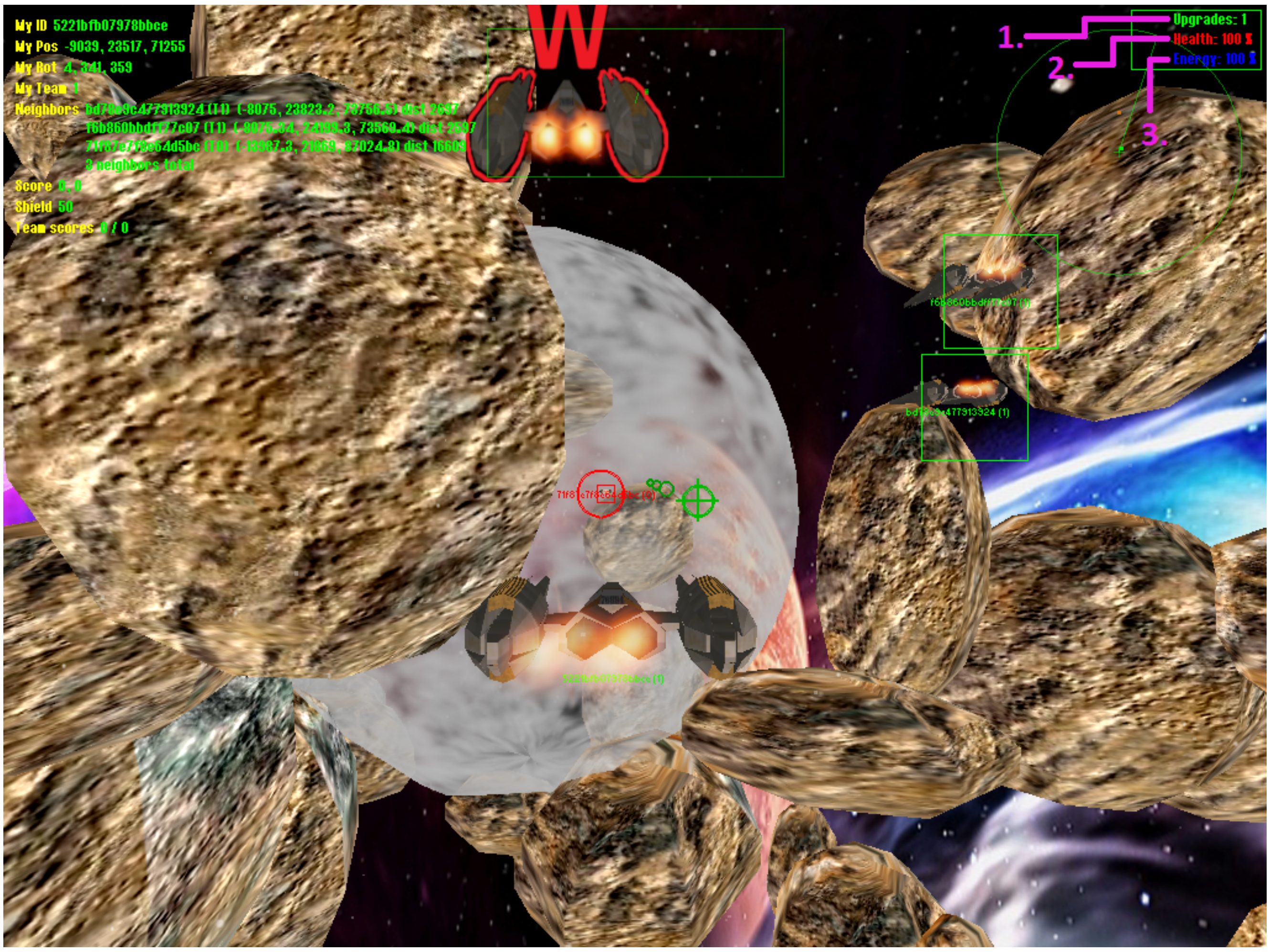

```
My ID  5221bfb07978bbce
My Pos  -9039, 23517, 71255
My Rot  4, 341, 359
My Team  1
Neighbors  bd78e9c477913924 (T1)  (-8075, 23823.2, 73756.5) dist 2697
           f6b860bbdff77c07 (T1)  (-8075.34, 24199.3, 73569.4) dist 2597
           71f87e7f8e64d5bc (T0)  (-13987.3, 21869, 87024.8) dist 16609
           3 neighbors total
Score  0, 0
Shield  50
Team scores  0 / 0
```
```
Upgrades: 1
Health: 100 %
Energy: 100 %
```

Figure 1: Gameplay of Planet π4

3. Energy: used to fire the weapons of the ship. If the ship has no energy, then its weapons cannot be fired. Energy is shown in an interval between 0% to 100%. Every ship recovers energy up to 100% as long the ship is not shooting or boosting.

Players joins a teams in order to play the game. With the current build of the game every team can have unlimited number of players. The number of teams in the game is also unlimited.

The main objective is still in development, but in this thesis the main objective of the game is to obtain as many kills as possible in a specific amount of time. When the time is up, the team with the highest number of kills wins the match.

The game also contains bonuses that the player can use in order to improve his/her chance to win the game. These bonuses are:

1. Shield Generator

2. Upgrade Points

Shield generators can fully restore the shield of the ship, which is the health measure of the game; a ship with no shield is as good as dead. Shield generators are disperse around the asteroid field, so it is easy to find one.

An upgrade point is where players can upgrade their ships. When an upgrade point is taken by a team, the next time the player respawns, the ship will be upgraded by the number of upgrade points that the team has under control. The more upgrade points the team has, the better the upgrade will be. Upgrades improve three things:

1. Energy (5% for each upgrade): More energy means that more bullets can be fired before reaching 0% of the energy bar.
2. Speed of the ship: More speed means that the team can flee and reach other places faster.
3. Shield (5% for each upgrade): More shield means a better chance to come alive after a fight.

More energy means more bullets can be fired, before reaching 0% of the energy bar, more speed means, that the team can flee faster and reach other places faster and more shield means a better chance to come alive after a fight.

These upgrade points are used in planet $\pi 4$ as points of interest. Every team in the game will like to have control over these upgrade points in order to outrun the competition.

2.1.3 Planet $\pi 4$ over peer-to-peer

Having Planet $\pi 4$ working with thousands of players over a peer-to-peer network is the goal of this project. In order to achieve this goal Planet $\pi 4$ uses different peer-to-peer approaches. The currently implemented approaches [21] in Planet $\pi 4$ are based on psense [2] and BubbleStorm [22]. The game also supports client-sever system, but it is only for testing proposes.

This game uses a simulator to simulate a peer-to-peer network. The simulator works with the three approaches mentioned before and it is used in this thesis to test the implemented AIs and the game itself.

2.2 AI State of art

In this section, the current state of art of AI for computer games is explained, showing different systems that are used in the industry of games. Also it will explain some advantages and disadvantages of these systems.

Over the last decade, game companies used different AI systems to create their games. Most of these systems were Rule Based System (RBS)[30], Finite State Machines (FSM)[30], Goal Oriented Action Planning (GOAP)[23], Hierarchical Task Network (HTN)[15] and Behavior Trees (BTs)[8].

2.2.1 Rule-based systems (RBS)

RBS AIs are based on rules. Basically if the condition of the rule is achieved then the bot, which is the agent, controlled by the AI, will react doing some action. This action is based on the condition of the rule, i.e. if the condition of the rule is have low shields, then a possible action could be go to the nearest shield generator.

This system is popular since is easy to implement. So it is preferred for games where the AI does not require complex actions. Unfortunately, the system is not perfect since the rules are only written in if-then capsules, which decrease the possibilities and actions that a bot can do. Another problem is the predefined behavior; this implies that the bot will do the same actions under the same situations. This could be very repetitive for a player and also gives the player the possibility to counter the opponent with the same tactic.

Popular games like Baldur Gate[4] and Virtua Fighters 2 [25] are implemented using RBS.

2.2.2 Finite-state machines (FSM)

Comparing FSM to RBS, FSM comes as a better candidate in terms of making a complex AI. The reason behind this is that FSM does not use if-then capsules to define its behavior but rather it uses states.

When working with states, each state decides the next state to follow, only when certain preconditions are met. This also means the state is responsible for doing the action that it is in charge of. For instance if the current state of the bot is Find Enemy then this state is responsible for finding a new enemy, it will do that until it finds one. When it finally finds one, the state will choose the next state to follow. This could be i.e. go to target if the target is far away and the bot needs to get near or it could be to engage the enemy, in the latter case the bot will engage the enemy in combat until the enemy is dead or the bot is dead.

This system is also easy and fast to implement. Problems usually occur when the behavior of the bot becomes too complex. This is a problem since for each new action that the bot requires a new state is also required. Than the more complex the AI grows, the larger and complicated the state machine becomes.

Another common problem with FSM is that the behavior of the bot is hard coded. This means that the bot will behave exactly as it was defined in the states. With a predetermined behavior, the player will notice this pattern and can predict it in the future, making it not suitable for replaying.

Popular games that implement this system in their games are Unreal [18] and Halflife [10].

2.2.3 Goal-oriented action planning (GOAP)

GOAP was invented by Jeff Orking [23] when he was working at Monolith developing F.E.A.R. [5] (First Encounter Assault Recon), one of the best games of 2005, with an innovating AI and with a terrific horror atmosphere. As Jeff Orking mentions in his paper [23], GOAP is a simplified version of the Stanford Research Institute Problem Solver (STRIPS), which is a method that generates plans to resolve problems. GOAP is oriented for computer games.

Nowadays many popular games use this system, in order to make their AIs. Here are some examples Fallout 3 [26], F.E.A.R. 2: Project Origin [6], Silent Hill: Homecoming [13], etc.

Looking at the two last systems, the predetermined behavior was a problem in terms of replay value. With GOAP system, this is no longer a problem. GOAP is goal oriented, which means that the AI is going to work its way to accomplish its current goal. Here is an example how GOAP works:

Imagine that the bot is controlling a space ship. The ship is in perfect status and is ready to fight. Seeing it current status, the bot decides that it's a good time to attack an enemy, so it decides to put its primary goal "attack near enemy". In GOAP all goals have preconditions and effects:

- **Goal**: "Attack near enemy".
- **Preconditions**: "Enemy found".
- **Effects**: "Enemy dead".

To fulfill the current precondition the bot must find a goal with the effect enemy found. In order to do this GOAP uses a planner that generates plans to satisfy the goal precondition. In this case the planner could give the bot a plan like this:

- **Goal**: “Set target near enemy”.
- **Preconditions**: none.
- **Effects**: “Enemy found”.

In this case the plan only contains a single goal that fulfills the primary goal precondition. But it could also be a very complicated plan with several goals to achieve.

This is how GOAP basically works: the bot has a primary goal and depending on the goal, it makes an appropriate plan to follow. This plan will contain several sub goals that will help the bot to reach its primary goal. If the bot encounters a problem in its plan, it will rethink the plan, making the behavior of the bot more versatile.

Implementing GOAP requires a lot of time and effort, which many companies do not want to invest. Also if GOAP has to solve a complicated problem, this will require more processing time and in terms of games, especially online multiplayer games, most companies only allocate reserved amount of processing time for AI.

2.2.4 Hierarchical task network (HTN)

HTN uses a planner in order to create plans to follow. HTN uses 3 different types of tasks: primitive, compound and goal tasks, which are necessary to accomplish the primary task. Primitive tasks can be executed immediately, without requiring some input or event. An example of a primitive task could be seek position, in this case the bot already knows the position, so it only uses the controls to reach this position.

The second important task in HTN is a compound task, this task cannot be done immediately and it requires accomplishing more than one primitive task in order to be successfully achieved. For instance if the bot requires going to the nearest shield generator in the game, it will require the following tasks:

- Find the nearest shield generator and set it as destination.
- Seek destination.

In order to complete the task go to the nearest shield generator, the two primitive tasks should be first accomplished. Also the order, in which the primitive task runs, plays a role. If we were to seek a destination without having a destination, we could end in breaking the game or having the bot doing a strange behavior.

The last task to follow is goal task, this task define which plan the bot has to follow, for instance if the goal of the bot is to win the game, the bot will generate a plan to accomplish this task.

Killzone 2 [14] is a fine example that implements HTN for AI.

2.2.5 Behavior Trees (BTs)

BTs have become more popular in the last years since games like halo 3 [28] and spore [12] implemented this system in order to create their AI.

BTs are goal oriented, making each node of the tree a goal to fulfill. Goals could be either primitive or composite.

Primitive goals are processed at once since they are easy to put in a single action or to not handle many events. On the other hand, composite goals don't only handle goals that are more complex with sub goals, it also checks:

1. In which order the sub-goals have to be fulfilled.
2. Depending on the situation, it chooses the most appropriate goals.

Example:

- **Goal**: "Don't be hungry"
- **Subgoals**: "Eat Food", "Go to kitchen" and "Get food"

In this case the composite goal will check its current state in the world. If it is already in the kitchen, it doesnâĂŹt need to go there and if it already got food, it only needs to eat it. But if the bot is not in the kitchen and does not have food, it will try to fulfill these subgoals in the following order:. First "Go to kitchen" then "Get food" and last but not least "Eat Food".

- **Goal**: "Don't be hungry"
- **Subgoals**: "Eat pizza", "Eat ice" and "Eat rice"

In this case the bot has different choices to make in order to fulfill its hunger goal, any of this sub goals will change its current state. So depending on where the bot is, it could decide which is the best option, for instance if the bot is near a pizzeria it will buy a pizza instead of looking for an ice.

BT is a very simple system but it has a lot of potential to be exploited in the coming years. It may not be as sophisticated as HTN and GOAP, but it requires lower processing time and it is also not too complex to implement.

3 Concept

With the information gathered in section 2 the following AIs are candidates to be implemented in this thesis: RBS, FSM, GOAP, HTN and BT. In order to choose which AI is the most appropriated, the AI must accomplish the following parameters:

- The AI requires low processing time.
- The AI system is flexible. Easy to tune.
- Developing the AI is not too complex. A complex AI requires more man power to develop.

AIs that require planners (GOAP and HTN) to create their strategies, need higher processing time than the other AIs and they are more complex to implement. RBS requires low processing time and its not complex to develop but as explained in section 2.2.1 this AI is not flexible enough. This leaves FSM and BT as candidates, both fulfil the conditions set above; therefore both AIs are capable of achieving the goal of this thesis. In order to evaluate the efficiency of the AI, this thesis will create two different

AI	Process Time	Flexibility	Dev. Complexity
RBS	Low	Low	Low
FSM	Low	Average	Average
GOAP	High	High	High
HTN	High	High	High
BT	Low	High	Average

Table 1: AIs comparison.

AIs. The first AI will be a FSM and it will be focused on attacking the enemies. The second AI will be implemented with BT and it will include the following features:

- Goals: The bot has always a goal. This goal tells the bot what to do.
- Goal Evaluators: Evaluators decides which goal is the most appropriate to use in certain situations.
- Chat: Reporting position of enemies is done via chat.
- Bonuses: The bot makes good use of all bonuses in the game.

4 Implementation

In this chapter the implementation of the AIs is described. First of all the used AI architecture is presented. Further on, the implemented AI controls are explained. The chapter ends presenting the implementation of the FSM AI and the goal oriented BT AI introduced in sections 2.2.2 and 2.2.5.

4.1 AI Architecture

The primary goal of our AI is to simulate human behaviour thus making the bots smart enough to create a challenge to real human players. In order to fulfil this goal, the AI should make the right decisions at the right time. This means that the AI should continuously check the current state of its world and based on that information make a smart action.

The second goal of our AI is to use the game chat to give information about the current state of the world. For instance if the bot sees an enemy, the bot will inform the team about the position of the enemy via chat. After the message arrives to other team members, they will know about the last position the enemy has been seen. This kind of tactics provides the team with a better perspective of the game world.

Taking into account the two goals above, the following architecture decision were taken:

- As the main game is coded in C++ , the AI is also coded in C++.
- The game engine is Irrlicht [16], so many functions used in this thesis are implemented by Irrlicht. From now on irr::... will denote an Irrlicht object.
- The AI is mostly implemented with interfaces. The advantage of using this design is that it facilitates the exchange of functionality between objects.
- Planet π4 uses a task system to manage sub processes in the game. In order to be part of the game, the AI has to inherit the class *ITask*, which is the interface in charge of the task in the game. When inheriting this class the following method has to be implemented:
  - **void executeTask()** which is called once for each frame of the game, leading to execution of the logic implemented in that method.
- The AI uses the same controls as a human player. This is mainly done by using the interface *IShipControl*.
- In order to be recognized as an AI class in planet π4, the AI class must inherit the *IBot* class, and implement the appropriate methods from its parent class.
- This thesis works with two different AI approaches based on Mat Buckland book [7]:
  1. **FSM AI:** The AI I implemented first, based on FSM architecture explained in section 2.2.2.
  2. **Goal Oriented BT AI:** This is the second implemented AI. This one uses BT to control bots and it is goal orient. This AI was explained in section 2.2.5.

4.2 AI controls

The controls used by the AI are the same available to human players. This was made possible for the AI by implementing the *IShipControl* interface. This interface contains methods to control the ship and also to give information about the current state of the ship. To be more specific the following methods are included in *IShipControl*:

void fire(bool isOn)
Sets the weapon system of the ship on , if *isOn* is true, otherwise turn the weapon system off.

int getTeamUpgradePointCount()
Gets the current number of upgrade points that the team has under control.

float getWeaponEnergy()
Gets the current available percentage of the ship's energy. The energy depends on the number of upgrade points that a team control. The more upgrade points the team currently has under control the more energy a ship will have. Each upgrade point increases the available energy by 5%.

void boost(bool pIsOn)
Sets boost active if *pIsOn* is *true*, otherwise sets the boost off. Boost increases drastically the speed of the ship.

bool getIsBoostActive()
Gets the current status of Boost. Returns *true* if boost is active, otherwise it returns *false*.

void accelerate(irr::f32 factor)
Sets the absolute speed of the ship; the range of *factor* is between [-1.0, 1.0]. Negative values mean that the ship will go backwards.

void slide(irr::f32 factor)
Controls the ship lateral slide. It works similar as *accelerate*, *factor* must be between [-1.0, 1.0]. A negative value means to slide laterally to the left side and a positive value means to slide to the right side.

void steerLeftRight(irr::f32 factor)
Controls the ship horizontal steering. It also has *factor* between [-1.0, 1.0]. A negative value means to turn left, a positive value means to turn right and a value equal to zero means that the ship shall stay centered. The closer the value of *factor* is to 1.0 or -1.0 the faster the ship will turn to the corresponding direction.

void steerUpDown(irr::f32 factor)
Controls the ship vertical steering. *SteerUpDown* works similar to *steerLeftRight*, with the difference that the ship will steer vertically instead of horizontally. Positive values will steer the ship upwards, negative values will steer the ship downwards and 0 will maintain the ship centered.

void lockOnTarget(IShip* pTargetShip)
Locks on the target specified by *pTargetShip*. This method sets the aim of the ship at an enemy ship. This allows a very accurate shooting. When using *lockOnTarget*, the functions *steerLeftRight* and *steerUpDown* are disabled. This function is not only available for bots. It's also implemented in game for human players.

The interface *IShipControl* not only provides the method to controls the ship, but it also includes an interface, that is in charge of sending messages to other players in the game, *IMessageListener*:

void publishToTeam(std::string message)
PublishToTeam is in charge of multicasting the message given by the parameter message. This message will only be received by team members.

void publishToAOI(std::string message)
publishToAOI is in charge of multicasting *message* only to players in the area of interest (AoI). This means that all players that are in the surroundings of the current ship position will receive this message. Both enemies and allies will receive the message.

4.3 ExtBot

ExtBot is one of the core classes for the AI. It contains functions that a bot requires to do several actions, like flying to a specific point, or retrieving the information about the closest ship or closest shield generator.

The following functions are the most relevant in *ExtBot*:

void void goToTarget(IShipControl* myShipControl, irr::core::vector3df Coordinates)
goToTarget takes control of the ship via *myShipControl* and navigates the ship to the vector given by *Coordinates*.

POI_ShieldRegenerator* getNearestShieldGenerator(AI_Dragon_Bot* pBot)
getNearestShieldGenerator returns a pointer of the shield generator closest to the position of the bot. In order to get the position of the bot, *pBot* is necessary. If this parameter is not provided, and/or no shield generator is found, this function will return *NULL*.

POI_UpgradePoint* getNearestUpgradePointDestination(AI_Dragon_Bot* pBot)
getNearestUpgradePointDestination works similarly to *getNearestShieldGenerator*. It returns a pointer of the closest upgrade point. If no upgrade point is found, it returns a *NULL*.

RemoteShip* getNearTarget(AI_Dragon_Bot* pBot)
getNearTarget returns a pointer of the closest enemy ship. If no enemy ships are in the area the returned pointer will be *NULL*.

4.4 IBot

IBot is an interface that all bot have to implement in order to work with planet $\pi 4$ system. IBot contains the following pure virtual function:

void init (IGameStateView* state, IShipControl* shipControl, ITaskEngine* taskEngine, IRandom* random)
Responsible for the initialization of the bot. This function is pure virtual. In case of the FSM implementation, it initializes the FSM for the specific bot, like described in section 4.5

4.5 FSM AI

This was the first AI implementation for this thesis. It is based on a FSM. Using this system allows to create an AI behaviour which can be easily debugged.

A FSM requires the definition of bojh states and transitions between states. States are implemented via a common interface which has to be inherited by all the actual states that want to take part of the FSM AI. For the transitions, each state is in charge of defining which transitions are allowed from the specific state, and thus which states are the potential ones to follow.

The static design of the FSM is captured by the *state* interface, the set of *state* classes and the definition of the transitions between states which is included within each state. Each state also contains the logic to decide which specific transition shall be activated based on a set of parameters. The execution of the FSM and thus the flows of the states must also be implemented. For this reason the class *StateMachine* was created, which is in charge of the changing the current status of the FSM to the next state and which also initializes and starts the AI.

Now let's take a look in more detail at the different classes which the FSM AI is made of.

4.5.1 State

This is an interface that acts as a reference to new states. In this interface the following functions are pure virtual, meaning that every class that inherits this interface has to implement these functions by itself.

std::string handleTransition()
Every state has to handle the transitions by themselves; this is required to change the current state into another state. This is done by *handleTransition*. This function will check the current parameters that are required in order to change to a specific state. If the parameters are met the current state will change to the state which fulfils the transition conditions. If none of the conditions of all possible transitions are met, the current state will remain the same. The return value is a string, which will tell the next state to follow.

void handleBehavior()
This function has the following tasks:

1. Executes the core logic that the state is in charge for. For instance if the main function of a state is *goToShieldGenerator*, this function will handle all the subtasks needed in order to guide the ship to the nearby shield generator.

2. The execution of *handleBehavior* contributes to the definition of some parameters in *handleTransition*. These parameters allows to change the current state.

void startStateTimer()
This function stores the time in which this state is accessed.

void endStateTimer()
In contrast to *startStateTimer*, this function stores the time in which the state stops working or the current state changes to another one.

4.5.2 State Machine

The *StateMachine* class is responsible for running the whole FSM. These are the tasks of the *StateMachine* class:

- *StateMachine* is the only class with control over the current state *m_currentState* of the FSM.

- Running the *StateMachine* for the first time it will initialize m_currentState and all other states: *AI_GoToShieldRegeneratorState* , *AI_GoToTargetState* , *AI_ShootingState* and *AI_RestState*.

- In every frame of the game, *StateMachine* will call *handleBehavior* and *handleTransition* from *m_currentState*.

- StateMachine inherits functionality from the following interfaces: *IBot*, *IStateDictionary* and *ITask*.

Apart from implementing these interfaces, the *stateMachine* class contains the following functions:

void init(IGameStateView* state, IShipControl* shipControl, TaskEngine* taskEngine, IRandom* random)
This function initializes all bots. This will give the bots access to the controls of the ship given by *shipControl*. Each bot will also have a pointer to *IGameStateView*, which is the current state of the world given by *state*, *taskEngine* allows the bot to subscribe its task to the engine. . And last but not least *random* parameter allows random behaviour for the bot.

void frame()
This function is called in every frame of the game and it is in charge of maintaining the FMS working.

4.5.3 Interfaces

IStateDictionary
IStateDictionary allows states to get information from other states. For instance if a state holds the information of the position of an enemy, it is possible to get this information via this interface, using *GetState(std::string pStateName)* and then asking the for the information required to the given state.

State* GetState(std::string pStateName)
This function will get the name of a state given by the parameter *pStateName*. Then it will return a pointer to the state with the same name.

4.5.4 Implemented states

In this chapter all states implemented in for the FSM are explained and displayed in **Figure 2**, including for each of them:

- Functionality: It explains what's done in the state.
- Transitions: Shows the possible transitions of this state.
- Reachable: Shows from which states is this state reachable.

AI_GoToShieldRegeneratorState

Functionality
This state is in charge of finding the nearest shield generator and navigating the ship to get there.

Transition

1. If the shield is not fully regenerated, then stay in the same state.
2. If the shield is fully regenerated, then change state to *AI_NewTargetState*.

Reachable
This state is reachable by all states. When the ship is in danger of being eliminated (Shields are under 30% of the maximum) it will change to this state and flee to the nearest shield generator.

AI_NewTargetState

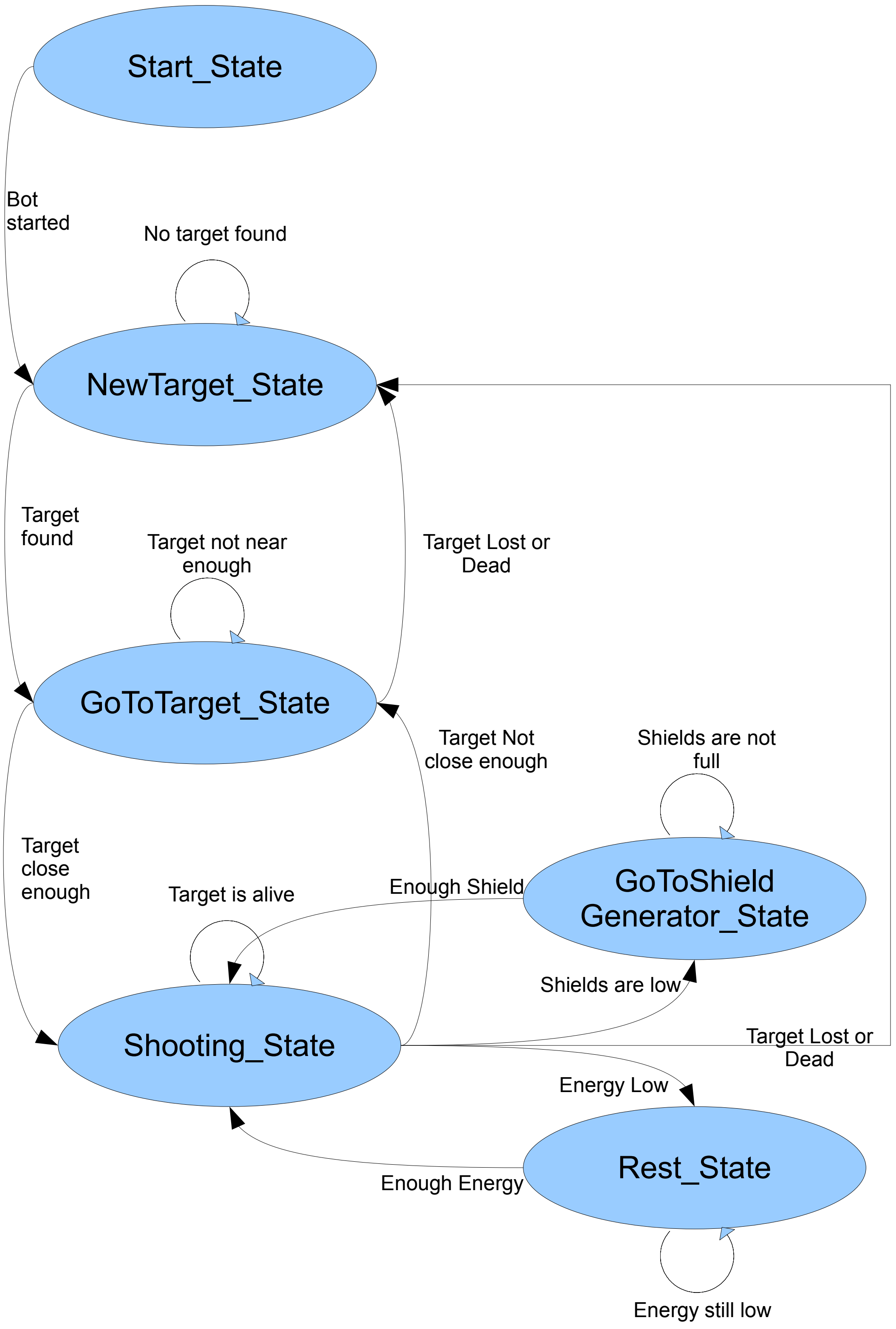


Figure 2: Graph of the state machine used for the FSM AI

Looking for a new nearby target is the objective of this state. It will find the closest enemy and set it as target.

Transition

1. Until no target is found, then stay in the same state.
2. If target is found and successfully stored, then change state to *GoToTarget_State*.
3. If the ship has less than 30% of the shield, go to *AI_GoToShieldRegeneratorState*.

Reachable
AI_NewTargetState can be reached by *AI_GoToShieldRegeneratorState* after the ship has fully charged shields. It can also be reached after finishing an enemy target in *AI_ShootingState*.

AI_GoToTargetState

This state will control and navigate the ship to a nearby enemy ship, for this reason this state comes after setting a target in *AI_NewTargetState*.

Transition

1. If the enemy ship is not in range, the current state will not be change.
2. When the enemy ship is in range, the current state will be change to *AI_Shooting_State*.
3. If the ship has less than 30% of the shield, go to *AI_GoToShieldRegeneratorState*.

Reachable
This state can be reached by *AI_NewTargetState*, after finding an enemy target.

AI_ShootingState
Shooting state will take care of reducing the competition by killing enemy ships. This state will use the target set by *AI_NewTargetState* and it will try to defeat it by shooting at it.

Transition

1. If the target dies or leave the game the current state will be change to *AI_NewTargetState*.
2. If the ship is taking a beating and its shields are lower than a 30% of the maximum shield, then change current state to *AI_GoToShieldGenerator*.
3. When the energy guns are overheating for too much continuous fire, then change current state to *AI_RestState*.
4. If the ship has less than 30% of the shield, go to *AI_GoToShieldRegeneratorState*.

Reachable
AI_ShootingState can be reached only after a target has been set and the target is in shooting range. This means that this state is only reachable by *AI_GoToTargetState*.

AI_RestState
This use of this state is made in order to reload the energy guns. The ship cannot fire limitless and after an amount of time the energy weapons start overheating, which will reduce the fire rate of the ship. This state will lock the guns until the guns recharge all the energy.

Transition

1. If guns are not fully recharged, stay in this state.
2. When guns are fully recharged, the return to *AI_Shooting_State*.
3. If the ship has less than 30% of the shield, go to *AI_GoToShieldRegeneratorState*.

Reachable
AI_RestState it can be only reach after the weapons systems of the ship are overheated, the weapons are only fired *Shooting_State*, which means that *AI_RestState* it only reachable from *Shooting_State*.

4.6 Goal Oriented AI

As explained in section 4.1 the second AI which was implemented for this game is goal oriented AI with BT. In order to run this AI in Planet $\pi 4$, the game has to initialize the class *AI_Dragon_Bot*. This class contains the "brain" of the bot and also runs the basic functions that the bot requires in the game.

4.6.1 AI_Dragon_Bot

The class *AI_Dragon_Bot* is the main class of the goal oriented AI. This class is in charge of evading obstacles, shooting enemy targets and receiving messages from team mates. This class also contains a "brain". The "brain" is the main function that decides which goal is the most relevant to do at certain time and situation.

In order to fulfil these actions *AI_Dragon_Bot* contains different functions. These functions will be called in every frame of the game in order to ensure a smooth behaviour from the AI. The following functions are implemented in this class:

void Update()
Update is the main function of this class which is invoked by the game in every frame of the game. This function in turns invokes *TakeAimAndShoot* and *EvadeObstacleAndThink*.

void TakeAimAndShoot()
TakeAimAndShoot aims at targets that are in vision range and shoot them. This function is invoked every time, meaning that the bot is going to shoot targets (if they are in vision range) regardless of the current goal.

void EvadeObstacleAndThink()
EvadeObstacleAndThink is in charge of evading obstacles such as Asteroids. Since evading asteroids takes a higher priority than following the current goal, this function will always check every second if a collision is possible, and only if not it will then run the "brain" of the AI by invoking AI_Goal_Think to decide which is the next most appropriate action to do.

void onMessage(const std::string& message, const PlayerId& from)
onMessage reads the current flow of messages and saves it in a list. This list contains the message as string and the *playerId* (the ID correspond to the ship that sent it).

4.6.2 Goal

In this implementation there are two types of goals: primitive goals and composite goals. Primitive goals are implemented by the class *AI_Goal* and composite goals by *AI_Goal_Composite*. *AI_Goal_Composite* is also a derivate of *AI_Goal* which means it also has the functionality of *AI_Goal*.

AI_Goal

AI_Goal works with statuses. Every goal in the game has a status which defines the current state of the goal. Each goal can either be:

1. Active: A goal is active when the bot is currently working on it.
2. Failed: Usually occurs when some unexpected behaviour occurs
3. Completed: When the current goal has been achieved by the bot.

The status of goals is handled via several functions, some of which are implemented in the *AI_Goal* class itself. Classes inheriting *AI_Goal* can use the functions below without implementing them again:

bool isActive()
Returns true if the current goal is active , false if not.

bool isInActive()
This functions is exactly the opposite of *isActive*, it will return false when the current goal is active, true if not.

bool isComplete()
Returns true if the current goal is completed, false if not.

bool hasFailed()
Returns true if the current goal is failed, false if not.

int GetType()
Returns an integer that correspondent to the following numbers:

- 1 = active
- 2 = failed
- 3 = completed

The following methods in *AI_Goal* are pure virtual, meaning that classes inheriting classes must define their own implementation:

void Active()
When a goal is activated for the first time, this function will be called. Different parameters that are required for the goal will be here initialized.

int Process()
Process() it is called by a bot on the goal currently set as a current one. The main functionality of the goal should be implemented in this method. This function will also return an integer describing its status after the of execution of *Process*: active, failed or completed.

void Terminate()
Every time a goal has reached its final goal, this method will be called. This allows the goal to finish and/or to terminate current tasks that are still running. For instance if the current goal of the ship is to reach an specific location in space, the Terminate() function will set the current speed of the ship to zero when the target location is reached.

AI_Goal_Composite

AI_Goal_Composite is also an *AI_Goal* inheriting all functionality of AI_Goal. In addition to the functions defined by *AI_Goal*, this class contains a list and three new functions to define and handle sub-goals:

std::list<AI_Goal*> *m_SubGoals*
m_SubGoals stores all the sub-goals that must be completed in order to fulfil the composite goal. When all sub-goals are and completed terminated this list will be empty, meaning that there are no more sub-goals to be done.

void AddSubGoal(AI_Goal* g)
AddSubGoal will put the given goal *AI_Goal* in the list *m_SubGoals*. Keep in mind that *AI_Goal* can be either a primitive goal or a composite goal.

int ProcessSubGoals()
ProcessSubGoals works as following:

1. Take the first goal from *m_SubGoals*.
2. Process this goal.
3. When the goal is in status *completed*, checks if there are more goals to be done in *m_SubGoals*.
4. If no more goals are to be done, then set the current goal status to Completed. Else it deletes the current goal from *m_SubGoals* and repeat procedure from (1).

void RemoveAllSubgoals()
RemoveAllSubgoals removes all the goals that are in *m_SubGoals*. This is usually called after a composite goal fails, since the goal will try to start from the beginning and it won't require the old goals in the list.

4.6.3 Implemented Goals

This subchapter explains the behaviour of the goals that are implemented in the AI. It will show the input that is required to initialize a goal, the actions that are done by the current goal and the conditions required to terminate the current goal.

Each of the goals described below takes as input a pointer of *AI_Dragon_Bot*. This is required in order for the goal to have access to the controls of the ship and have information about the game world.

Goal	Type	Sub-Goals
Seek to Position	Primitive	-
Hunt Enemy	Primitive	-
Explore	Primitive	-
SetNearEnemy	Primitive	-
KillNearEnemy	Composite	SetNearEnemy / Hunt Enemy
GoToShieldRegen	Composite	Seek to Position
CaptureUpgradePoint	Composite	Seek to Position

Table 2: Primitive and Composite Goals.

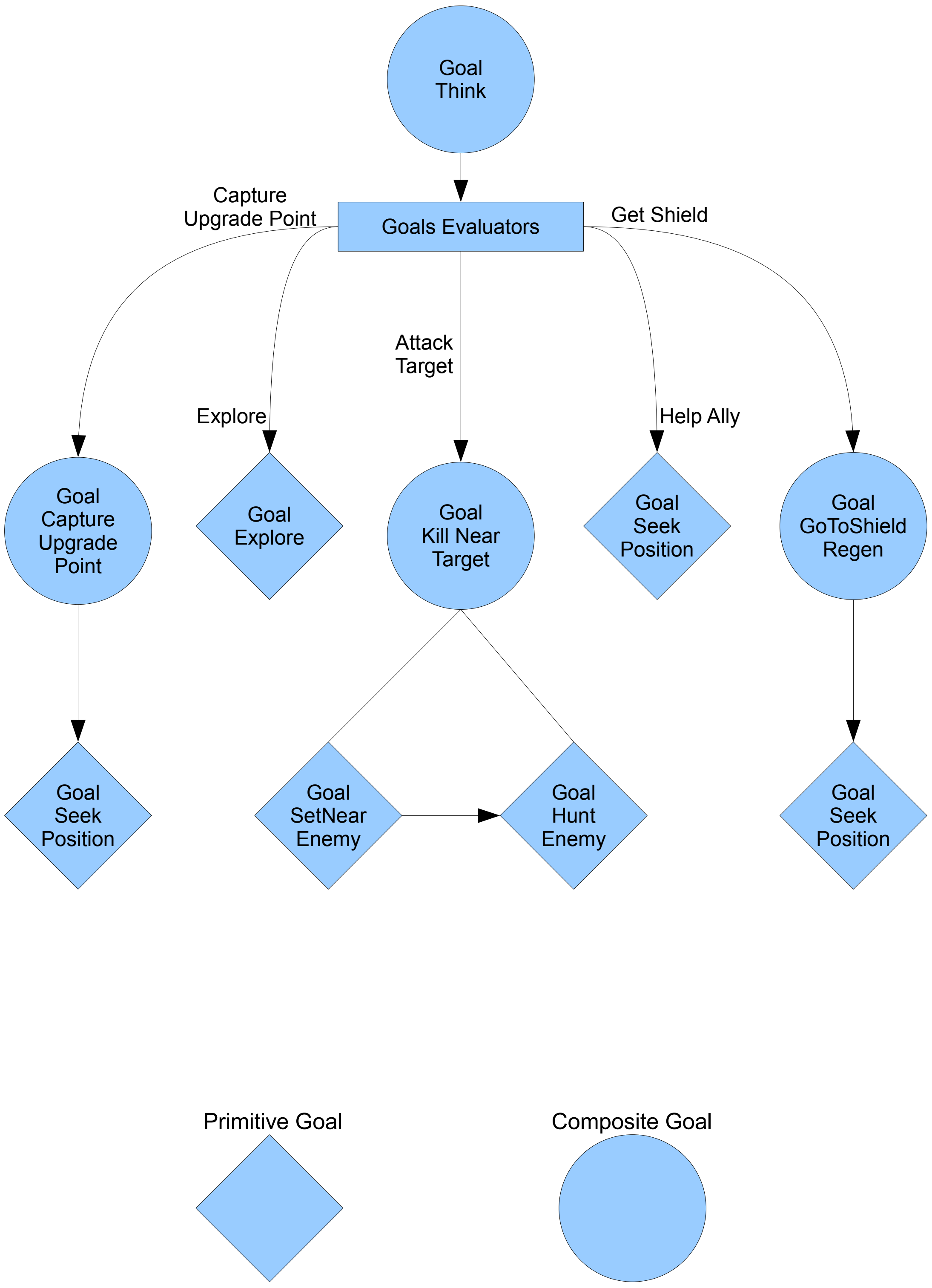


Figure 3: Graph from the implemented BT AI.

4.6.4 Primitive Goals

AI_Goal_SeekToPosition

Input
This primitive goal contains a vector (irr::core::3dfvector) holding the position where the bot should go. It also contains a float number, which defines the minimum distance from target position that the bot must achieve to complete the goal. For instance if this float number is 500 the bot will continue to move towards the destination, until the distance between the bot and destination is less then 500. This value is stored in *m_reachDistance*.

Process
This goal will activate the boost of the ship and set the current speed to its maximum (speed = 1.0). Also it will steer the ship in direction towards the destination.

Termination
This goal is completed when:

$$DistanceBetween(CPos, DPos) < distance \tag{1}$$

CPos: Current Position of the bot.
DPos: Destination position.
distance: Minimum distance set by m_reachDistance.

It could also be terminated when the bot is destroyed.

AI_Goal_HuntEnemy

Input
AI_Goal_HuntEnemy takes as input only an instance of *AI_Dragon_Bot*. This goal requires the information contained in *m_CurrentEnemyTarget*, in order to access the position of the target.

Process
This goal uses different tactics in order complete its job. The main idea is to follow the enemy target and to take it down. Since it is not always wise to follow the target, this goal implements different strategies depending on the distance between the target and the bot ship. The strategies are following:

1. If the target is far away and it is not in reach of bullets, than get closer to the target at full speed (Speed = 1.0 and Boost = true).
2. If the target is in bullet range, deactivate boost engines and shoot the target. Also try to follow the target in the mean time. (Speed = 1.0 and Boost = false).
3. If the target is too close try to flank it. This will active the lateral thrust of the ship and the bot will flight around the enemy ship. Also keep shooting at the target. (Speed = 1.0 Boost = false and SteerLeftRight = 1.0).

This will ensure that the bot will stay in combat as long as the enemy is alive or the bot dies.

Termination
The only ways to terminate this goal are either that the bot dies or the target enemy dies.

AI_Goal_SetNearEnemy

Input
AI_Goal_SetNearEnemy is a primitive goal and it only requires as input an *AI_Dragon_Bot* pointer.

Process
This goal looks at the current position of the bot and also looks for enemies which are in the area. This may find several targets, but only the one closest to the bot is selected. In order to do this the variable *m_CurrentEnemyTarget* will be set with the found enemy. In order to prevent problems with old and new information about the target, *m_ CurrentEnemyTarget* is set to NULL at the beginning of the process, every time this goal is called. When the target is NULL, it means that there are no current targets available.

Termination
To terminate this goal, an enemy must be found. So as long as *m_CurrentEnemyTarget* is NULL, the goal is not terminated. When *m_CurrentEnemyTarget* is successfully replaced with a target, then the goal will be terminated. This goal could also be terminated in case the bot is destroyed.

4.6.5 Composite Goals

AI_Goal_KillNearEnemy

Input
AI_Goal_KillNearEnemy is a composite goal and it uses as input an instance of *AI_Dragon_Bot*. This goal uses as sub-goals *AI_Goal_HuntEnemy* and *AI_Goal_SetNearEnemy*.

Process
Process of this goal is relative simple. First add AI_Goal_HuntEnemy and AI_Goal_SetNearEnemy to the list of sub-goals *m_SubGoals* using AddSubGoal(AI_Goal* g). Then call ProcessSubGoals() in order to process all sub-goals. This will ensure that all sub-goals are processed in an appropriate way.

Termination
This goal can be terminated, when all sub-goals are completed or when the bot dies. It is also possible that the enemy escapes from the bot, in this case this goal will also be terminated.

AI_Goal_GoToShieldRegen

Input
AI_Goal_GoToShieldRegen needs only an instance of *AI_Dragon_Bot*. This goal uses *AI_Goal_SeekToPosition* as sub-goal.

Process
AI_Goal_GoToShieldRegen will compare the current position of the bot and check with nearby shield generators. After it finds which shield generator is the closest, this goal will add *AI_Goal_SeekToPosition* to the sub-goals with the position of the shield generator. Then *AI_Goal_SeekToPosition* will navigate the ship to the shield generator.

Termination
This goal will be terminated when the ship reaches its position and recovers the amount of shield that is desired.

AI_Goal_CaptureUpgradePoint

Input
This goal allows bots to capture different upgrade points around the map. In order to complete this goal *AI_Goal_SeekToPosition* is required.

Process
The main idea of this goal is to find the nearest upgrade point, preferable if the upgrade point is currently controlled by the enemy, and then go to this upgrade point and take it over.

Here is the process of this goal:

1. Process *AI_Goal_CaptureUpgradePoint* and get the closest enemy upgrade point, if the enemy upgrade point is too far away (enough not be visible in the game distance >= (Vision Range) of the bot), find the nearest free upgrade point.
2. After finding the upgrade point, process *AI_Goal_SeekToPosition*. This will navigate the ship to the upgrade point.

Termination
This goal can be terminated either by the bot reaching the upgrade point, by dying or when the upgrade point is captured. The bot will not wait for the upgrade point to be controlled, instead as soon the bot reached the upgrade point, this goal is terminated. Then the bot will choose a new goal, here there are two possibilities:

1. If there are no enemies in the area the goal *AI_Goal_CaptureUpgradePoint* will have the highest priority since the upgrade points is not controlled.
2. If there are enemies nearby, the goal *AI_Goal_KillNearEnemy* will take the highest priority.

AI_Goal_Explore

Input
The goal explore is primitive and allows the bot to explore the map. This will only happened if the bot got nothing better to do, meaning there are no enemies nearby or there are no upgrade points to capture. As input this goal only needs the instance of the bot.

Process
Here is the process of this goal:

1. If this goal is selected, then look at the current position of the ship.
2. If the ship position is not near the middle of the map (distance <= VisionRange), then go back to the middle of the map. Otherwise choose a random position to explore.

Termination
This goal is terminated when the bot reach the explore position.

4.6.6 Goal Evaluators

At any time, bots has to decide which of the goals shall be pursued. The metric for the decision is implemented by the goal evaluators. These evaluators will check the current conditions of the bots (position, health, etc...). After checking all conditions each evaluator will return a value between 0.0 and 1.0. This value represents the level of desirability of a goal, where 1.0 represents the most desirable and 0.0 the least. The evaluator with the highest desirability is in charge of setting the necessary sub-goals in *AI_GoalThink*. After all sub-goals are set, the function of the evaluator will be terminated. The process repeats each time a bot completes a goal or dies.

Goal Evaluator	Tweaker	Set Goal
Get Shield	1.0	GoToShieldRegen
Attack Target	0.9	KillNearEnemy
Capture Upgrade Point	0.9	CaptureUpgradePoint
Help Ally	0.5	SeekToPosition
Explore	0.0001	Explore

Table 3: Goal evaluators set the current goal when they get highest priority. The calculated priority was tuned via Tweaker to achieve smarter AI strategies

AI_GoalEvaluator_AttackTarget

Definition
This goal evaluator will calculate the desirability to attack a nearby target. In order to calculate this desirability, this evaluator takes in account how much life the bot has left and the current distance between the bot and the nearest enemy. If the bot has enough life and the enemy is near, then the desirability will be greater.

Formula

$$Desirability = Tweaker * (\frac{B_{CS}}{B_{MS}}) * (\frac{B_{VR}}{B_{CDT}}) \tag{2}$$

- **B**$_{CS}$: Bot current shield status
- **B**$_{MS}$: Bot maximum shield value
- **B**$_{VR}$: Bot vision range distance
- **B**$_{CDT}$: Bot current distance to enemy target
- Tweaker : The tweaker is value between (0.0 and 1.0). it is used by the developer of the AI to tune the result of the formula. For instance if the developer doesn't want this to have more than 50% of desirability at any time, he/she can achieve this by setting the tweaker to 0.5 without changing any other parameter in the equation. The tweaker mechanism is used in all goal evaluators.

If the distance to the target is too large (**B**$_{CDT}$ $>=$ 2***B**$_{VR}$) the desirability formula will not be used; instead the desirability will be zero.

AI_GoalEvaluator_GetShield

Definition

AI_GoalEvaluator_GetShield calculates the desirability of going to a nearby shield generator. This is an useful tactic if the bot recently fought an enemy and won that fight, but it was injured during the last combat. Here the bot has to check different conditions in order to calculate its desirability:

- First the distance of the shield generator, if it is too far away the desirability will be zero.
- Second the status of the shields. If they are full there is no need to regenerate them.
- After checking the first two conditions, another condition comes into play when calculating the desirability. If an enemy is too close to the bot position, it will get quickly killed when trying to escape. For these reason if an enemy is nearby the desirability will also be low. This will means that the bot will take a kamikaze stance and try to fight until it dies.

Formula

$$Desirability = Tweaker * (\frac{B_{MH} - B_{CH}}{B_{MH}}) * (\frac{B_{NSD}}{B_{CS}}) * (\frac{B_{NDT} - B_{CDT}}{B_{NDT}}) \tag{3}$$

- **B**$_{MH}$: Maximum value for shield.
- **B**$_{CH}$: Value containing the current amount left of shield.
- **B**$_{NSD}$: Optimal distance to a shield generator.
- **B**$_{CDTS}$: Current distance to a shield generator.
- **B**$_{NTD}$: Optimal distance to target.
- **B**$_{CDTT}$: Current distance to target.

AI_GoalEvaluator_CaptureUpgrade

Definition

This is one of the most important evaluators in the game since it defines the tactic that the whole team will take in order to have an advantage over other teams.

This evaluator will calculate the desirability of capturing an upgrade point. In order to make the bots more competitive the bots will prefer to take over a controlled upgrade point instead of taking a free one. This is desired because taking a controlled upgrade point will not only benefit the team but it will also take away the advantage that the other team had.

For calculating the desirability, the distance to the closest enemy upgrade point is calculated, the nearer to the bot position, the more desired it is. If no enemy upgrade points are taken, the bot will take the closest free upgrade point instead. Also if the enemy upgrade point is too far away (Distance between bot and nearest shield upgrade greater than 1000000) the bot will also take the closest free upgrade point, but if this one is also far away the desirability will be zero.

Since capturing an upgrade point usually means to combat enemies , the current shield of the bot is also an important requirement to capture an upgrade point. After the team has a specific number of upgrade points under control, this goal evaluator will become less relevant and instead the bot will concentrate more in combat.

Formula The resulting formula to account for all conditions described above is the following:

$$Desirability = Tweaker * (\frac{B_{CH}}{B_{MH}}) * (\frac{B_{UPOD}}{B_{CDUP}}) \tag{4}$$

- **B**$_{MH}$: Maximum value for shield.
- **B**$_{CH}$: Value containing the current amount left of shield.
- **B**$_{DtCS}$: Current distance to message target.
- **B**$_{MOD}$: Optimal distance to message target.

AI_GoalEvaluator_Help

The goal evaluator *AI_GoalEvaluator_Help* is essential for team play between bots. The main idea is to read the current messages from team mates, then after checking which of the messages is the most relevant, this evaluator will calculate the desirability of helping an ally. This desirability is mostly based on the distance between the bot and the ally. If the bot is not in the area of interest of the bot which means the area that the bot can see, the message will have less desirability. Otherwise if the message is in the area of interest of the bot it will become a relevant message and therefore the desirability will increase. Another important point is the current status of the shield of the bot: the bot will not be able to help an ally if the bot it is almost dying, for this reason the lower the shield of the bots the less desirable this goal is.

Formula

$$Desirability = Tweaker * (\frac{B_{CH}}{B_{MH}}) * (\frac{B_{MOD}}{B_{DtCM}}) \tag{5}$$

- **B**$_{MH}$: Maximum value for shield.
- **B**$_{CH}$: Value containing the current amount left of shield.
- **B**$_{DtCS}$: Distance to current message sender.
- **B**$_{MOD}$: Optimal distance to message target.

AI_GoalEvaluator_Explore

Definition

This goal evaluator is created in order to avoid idle behaviour. This idle behaviour usually happens when the bot calculates all evaluators and gets for every one of them a priority of zero, meaning that the bot has nothing interesting to do. In this case it is be useful for the team to explore the map and get information about the surroundings.

Formula

$$Desirability = Tweaker \tag{6}$$

The Tweaker for this goal evaluator is currently set to 0.0001. This evaluator is only relevant when the desirability of all other goal evaluators is zero.

4.6.7 Execution

Goals and goal evaluators define the pieces of the logic which makes up the strategy and intelligence of the AI. Both goals and goal evaluators are executed by the Ġoal Think.̣

Goal Think

Goal Think is also a composite goal and it work exactly like the other composite goals. However this goal as a specific role, it is like the “brain” of the bot, it invokes the goal evaluators and it decides which goal to schedule next based on the desirability results.

In addition to a standard composite goal, this class contains the following list of evaluators and functions:

std::list<AI_Goal_Evaluator*> m_Evaluators
This list contains all the evaluators that were defined earlier in this chapter. They are push in this list when the goal this initialized.

void Arbitrate()
Arbitrate will take a look in the list *m_Evaluators*, then it will call for every evaluator the function *CalculateDesirability(AI_Dragon_Bot* pBot)*, this will return the desirability for each evaluator. After checking which evaluator has the highest desirability(closest to 1.0), *Arbitrate* calls the function *SetGoal()* from the winner evaluator. *SetGoal()* calls the "brain" in *AI_Dragon_Bot* (which is always *AI_Goal_Think**) and sets the corresponding goal.

This function will be call every time the bot dies or when the current goal is finished.

void AddGoal_KillNearEnemy()
Add goal KillNearEnemy to the list *m_SubGoals*.

void AddGoal_GoToShieldRegen()
Add to the goal list m_subgoals the goal GoToShieldGenerator.

void AddGoal_CaptureUpgrade()
Add to the goal list the goal CaptureUpgrade.

void AddGoal_Explore()
This function will add the goal Explore to the list *m_SubGoals*.

5 Evaluation

This chapter presents the evaluation and analysis of the AIs described in sections 4.5 and 4.6. It begins with an overview of the considered metrics, followed by a description of the defined simulation scenarios and simulation environment. Further on, simulation results are presented and the game performance of the AIs is evaluated. The chapter ends by summarizing the most relevant results and attributes of each AI approach.

5.1 Evaluation Goal

The main goal of this chapter is to evaluate the two AIs that are implemented in this thesis. The AI with goal oriented BT is set to play more as a team. The other AI based on FSM one will concentrate more on attacking enemies.

5.2 Metrics

In order to evaluate the behaviour and the performance of the bots, metrics are required. These metrics evaluates how well the AI plays in the game.Here is what is going to be measured: **Performance game**

- Number of kills: This refers to the number of kills that a team has at the end of a simulation.
- Number of deaths: This will tell how many times.
- Number of upgrade points under control: This measure will tell how many upgrade points has a team under control at the end of the simulation.
- Kills-Deaths Ratio (K-D Ratio): Kills minus deaths shows how well a player performs during the game, if the K-D Ratio is positive means that the player or the team are getting more kills than deaths which in most games is a metric of good performance.

5.3 Simulation scenario

The following parameters were used in order to define each game simulation scenario:

- Number of teams: Depending of the number of teams, the game will become more fluid and lively. In the simulations the teams vary between 2 and 4 teams per game.
- Number of players: This also depends on the current test, but the number of bots per teams start from 3 bots until 10 bots. Keep in mind that every team has the same amount of players for all tests.
- Time: Time is a very important metric, in order to take a relevant value for time this thesis will take the average time that a player usually plays a MMOG. This time can vary between 70 to 120 min [29].
- Size of the map measured by total amount of upgrade points: This metric shows how big the map is, if the map is too big, then players will get lost and won't find any enemies or allies at all. The size of the map in the simulations varies between 25 and 36 upgrade points.
- AI: The tested AI depends on the current simulation. It could either be FSM AI or goal oriented BT.

Each simulation that was done corresponds to a specific simulation scenario. The following procedure shows how the simulation scenario is set: at the beginning of the simulation all peers will not start at the same time, this is because the simulator will add player after player, this usually takes approximate 30 seconds of game time, so if there are 10 bots in the game it will take approximate 5 minutes for all the bots to join the game. The simulation data will start when the first player enter the game. The configured number of teams is going to play the game for about 100 minutes of game and after that the simulation will be ended. Then all data collected for the simulation will be analysed later in this chapter.

5.4 Simulation Environment

The simulations are run via a simulator developed by [21]. The simulator can simulate Planet π4 games with several number of players. If the number of players and the CPU of the computer is powerful enough the simulation can be run in real time or even faster. Otherwise the simulation runs slower. The simulator has different network configurations. These configurations are: P-Sense, BubbleStorm and Client-Server. This thesis uses P-Sense as the main configuration since it has prove to be the most reliable. For all the simulations executed the following computer was used:

- Processor: Dual Core 2.1 GHz
- Ram: 2GB

5.5 Simulation results

In this chapter all the simulations results are displayed and explained.

These are the results of the several simulations I ran to test the performance of the bots in the game.

5.5.1 First Simulation

Simulation parameters:

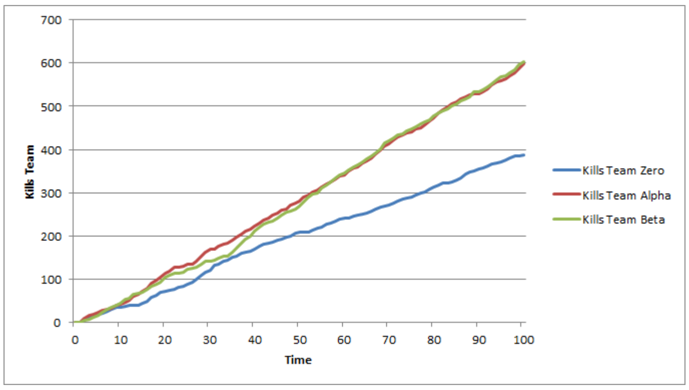
- AI: Goal oriented BT.
- Number of total players: 9.
- Number of players: per team 3.
- Number of teams: 3.
- Number of upgrade points in the level: 36.
- Simulation time: 100 min.
- Simulation system: P-Sense.

After analyzing the results of the simulation there are some interesting behaviours in the AI.

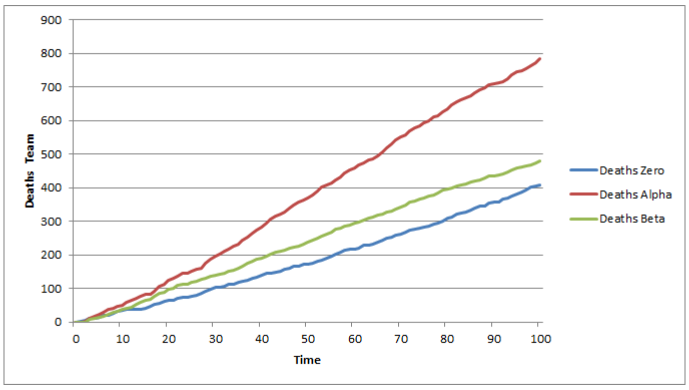
The game was won by team Beta with 603 kills followed by team Alpha with 598 kills. This means that only 5 kills made the difference between both teams. Team Zero only 388 kills in the end.

After looking closely to the information provided by the simulation, it can be conclude that:

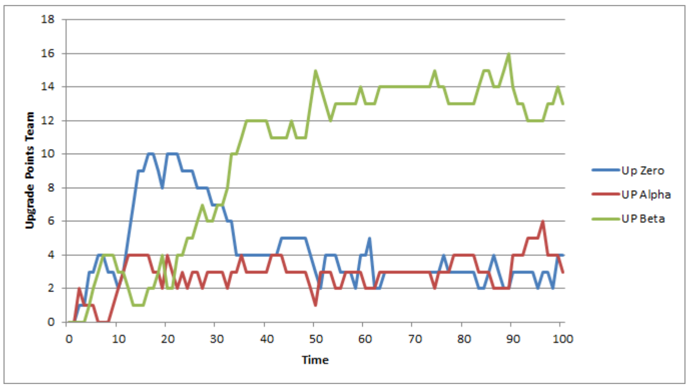
- Team Beta won the game with the highest amount of kills; although team Alpha almost won the game with a difference of 5 kills.
- Team Beta was very successful in controlling and defending most of the upgrade points in the level.
- Team Zero and Alpha were not successfully in capturing upgrade points, which resulted in dying a lot.
- Team Alpha is the team with most deaths.
- Team Zero is the team with fewer deaths.



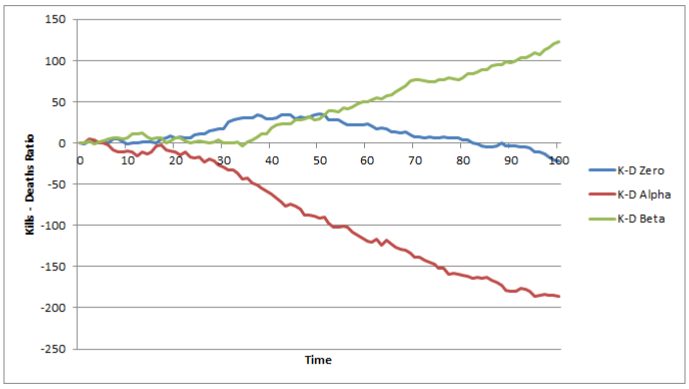
(a) Kills per team.



(b) Deaths per team.



(c) Upgrade points per team.



(d) Kills/Deaths ratio.

Figure 4: Simulation 1

- Team Beta has the best Kill/Death Ratio. When a player got a positive Kill/Death Ration it means that the player plays good and helps the team to achieve victory. On the other hand if a team player got a negative Kill/Death Ratio will reduce the chance of winning the game.
- Dying may be a bad thing in a game with 2 teams, because if one bot dies it means that the other team will receive a kill for it. But when playing with more than 2 teams, this rules changes a little bit. Dying a lot will not be too bad if the team receiving the kill is not the winning team.
- When a bot dies in the game it will respawn immediately in the middle of the level. This affects the behaviour of the bot since there is a high chance that a lot of players are in the middle of the level. This means that the bot has to fight immediately after it respawns. In the game when a player respawns he/she has 2 seconds of invincibility, meaning that no bullets will damage it and he/she can use this invincibility to his/her own advantage and makes a few shots before the invincibility wears off. Team Alpha not only take advantage of this feature, but also of the fact that all player respawns in the middle of the map, taking the opportunity to perform kills more easily after respawns, as the enemy teams usually don't have full energy and health at this point of time.
- Team Alpha got the most upgrade points in the first 30 minutes of the game as shown in **Figure 4(c)**. In this period of time Team Alpha reduced the amount of deaths over time. But the most relevant advantage obtained via upgrade points was shown by Team Beta. When Team Beta started to have control over most of the upgrade points, in the simulation after the minute 35, not only the amount of deaths over time was reduced, but also the amount of kills was increased to the point of catching up and surpassing Team Alpha as shown in **Figure 4(a)** and **Figure 4(b)**.

For this first test team Beta used a nice strategy to control most of the upgrade point, on the other hand team Alpha was not successful in claiming the upgrade points, but this benefit the team in the end since Alpha receives most of the kills from team zero. This was only possible for Alpha because of the current set of the respawn points.

5.5.2 Second Simulation

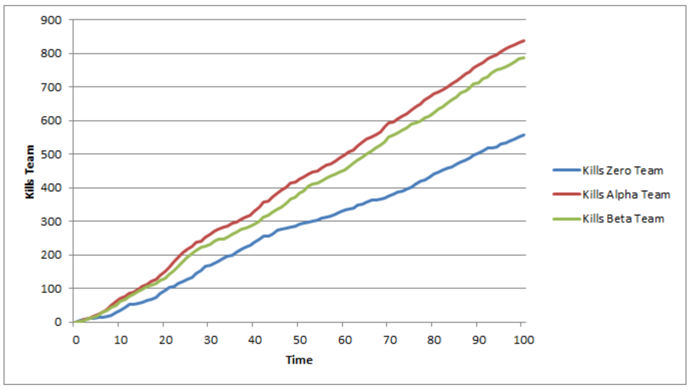
- AI: Goal oriented BT.
- Number of total players: 12.
- Number of players: per team 4.
- Number of teams: 3.
- Number of upgrade points in the level: 36.
- Simulation time: 100 min.
- Simulation system: P-Sense.

This simulation is very similar to the first simulation, but this time the number of players has increased in each team by one. More players mean more targets to kill but also all will be more exposed to fire than before.

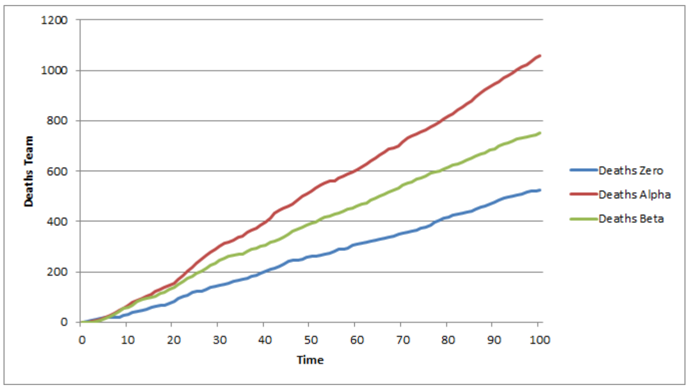
Taking a look at the information of this simulation, the following data shows some interesting behaviours:

- This time Alpha won the match using the same strategy of the first simulation.
- Team Zero got the most number of upgrade points in the game.
- Team Alpha got the most number of deaths and kills.
- Team Zero is the team with the highest average life time.
- Team Beta tries to regain control in the last 25 minutes of the game by controlling more upgrade points. This reduces the amount of deaths of the team but does not increase the amount of kills.

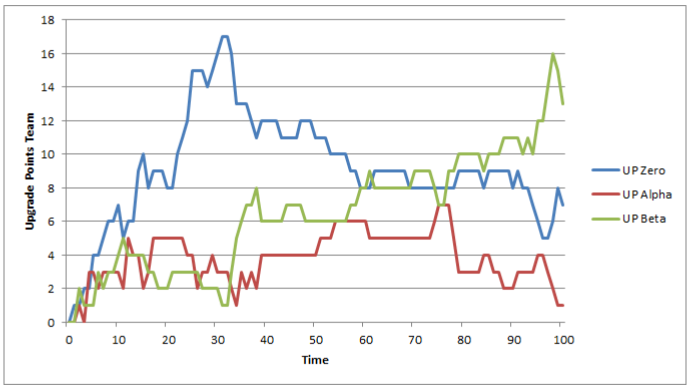
The second simulation shows that a kamikaze strategy prevails over a good strategy. Nevertheless this reckless strategy only works when there are more than 2 teams, as the results of the third simulation shows.



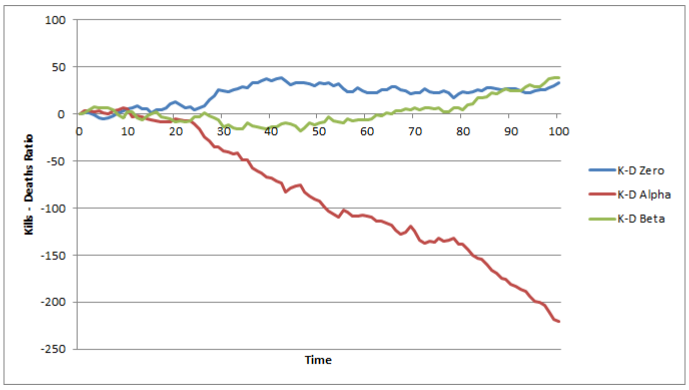
(a) Kills per team.



(b) Deaths per team.



(c) Upgrade points per team.



(d) Kills/Deaths ratio.

Figure 5: Simulation 2

5.5.3 Third Simulation

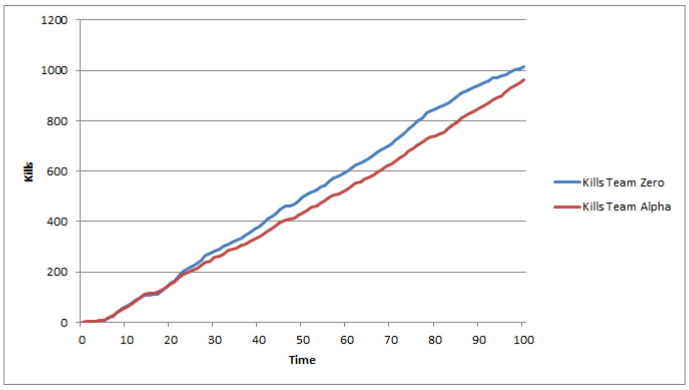
- AI: Goal oriented BT.
- Number of total players: 10.
- Number of players: per team 5.
- Number of teams: 2.
- Number of upgrade points in the level: 36.
- Simulation time: 100 min.
- Simulation system: P-Sense.

This time the simulation was run with only two teams each with 5 players. The rest of the setup is identical to previous simulations.

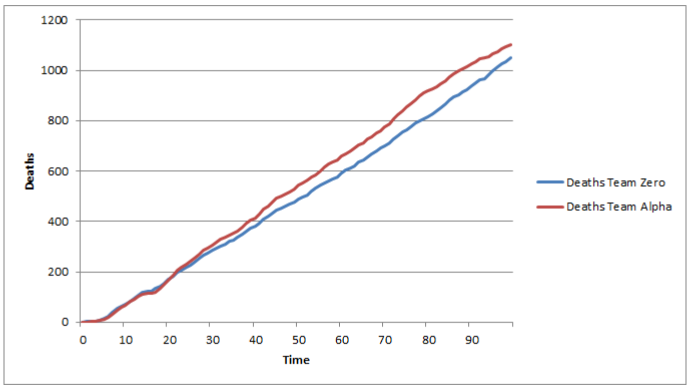
The following conclusions were made from the data that was gathered:

- Team Zero won the match with 58 kills more than team Alpha.
- Team Alpha has the highest number of deaths.
- Team Zero got most of the time the majority of upgrade points in the game.
- Team Zero got control of the game with a positive K-D ratio until Team Alpha started to regain control of the upgrade points.
- Team Zero increased the life time of the bots after taking more than 10 Upgrade points.
- With this data it can be concluded that in this specific setup upgrade points do decide the fate of the game. A team that controls more upgrade points than the opposing team will have a better chance to survive in combat.

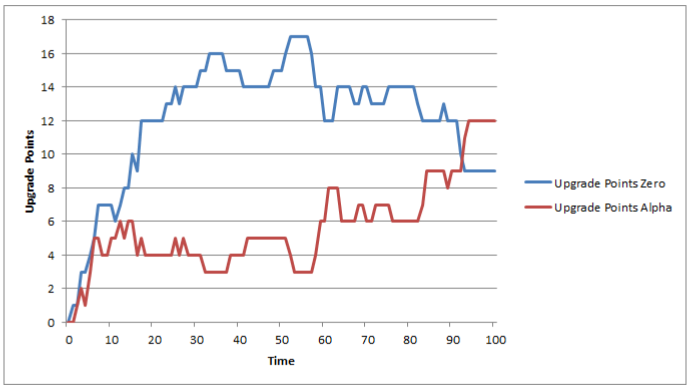
In this simulation the kamikaze strategy, which was successful in previous simulations, did not prevail over the good strategy. This time the team with the most upgrade points won the match, this shows that the upgrade points are vital in order to win a game against a single team, but it is not necessary the best strategy if there are a lot of teams.



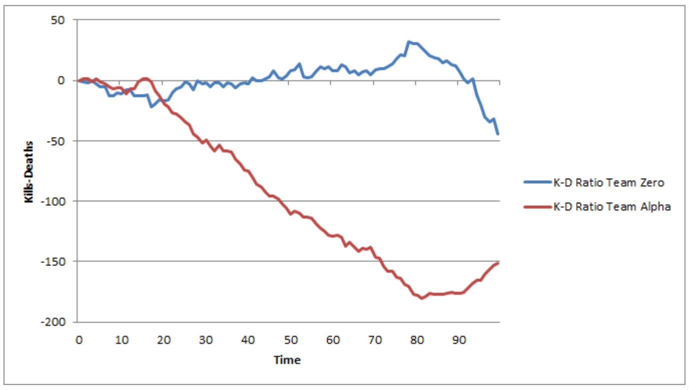
(a) Kills per team.



(b) Deaths per team.



(c) Upgrade points per team.



(d) Kills/Deaths ratio.

Figure 6: Simulation 3

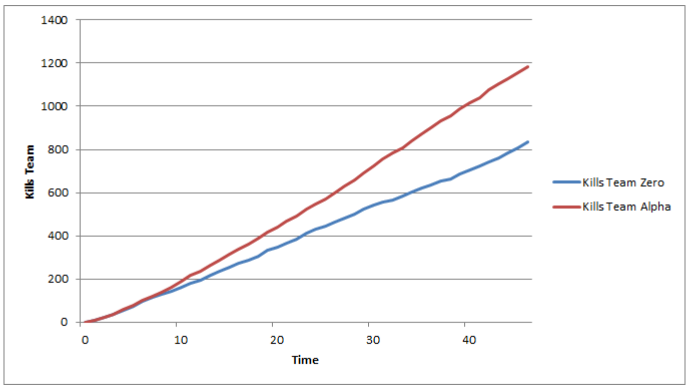
5.5.4 Fourth Simulation

- AI: Goal oriented BT.
- Number of total players: 20.
- Number of players: per team 10.
- Number of teams: 2.
- Number of upgrade points in the level: 25.
- Simulation time: 48 min.
- Simulation system: P-Sense.

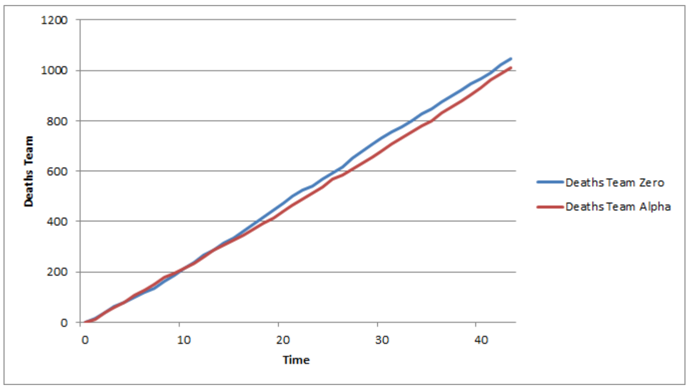
In this simulation the number of bots was set to 20. The simulation time was also set to 100 minutes but for technical reason related to the simulator, it was only possible to simulate the first 48 minutes. During this reduced timeframe it was already possible to observer some interesting results. Following conclusions were made from the data gathered:

- Team Alpha won the match with 348 kills more than team Zero as shown in **Figure 7(a)**.
- Team Zero only got 38 more deaths than team Alpha as shown in **Figure 7(b)**. This can be explained in two ways: either it was due to friendly fire or to colliding against a ship or an asteroid.
- Upgrade points increased greatly the amount of kills of team Alpha.
- The kills-deaths ratio was negative the first 20 minutes as **Figure 7(d)** shows.
- Team Zero got control of the game with a positive K-D ratio until Team Alpha started to regain control of the upgrade points.
- Team Alpha increased the life time of the bots after 10 minutes of game time. The reason for this change is that Team Alpha has more upgrade points than team Zero at that point.

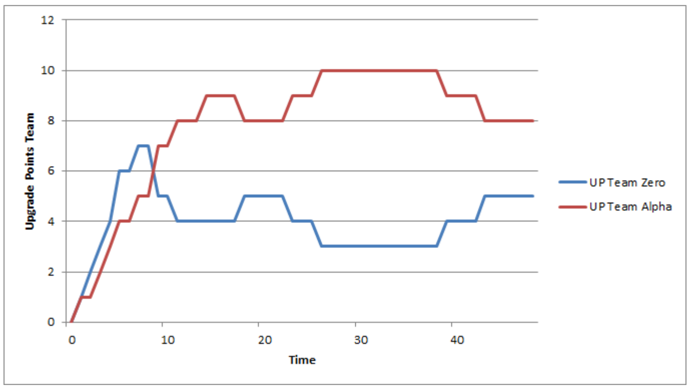
With this last simulation the number of bots plays an essential role in the number of deaths for both teams. The reason is that both teams got large armies at the middle of the map and fired bullets cannot differentiate between allies and enemies. More bots implies more bullets being shot. Also the conclusion made in simulation 3 applies in this simulation. The team that has control over the most upgrade points in the level, has a higher chance of winning the game.



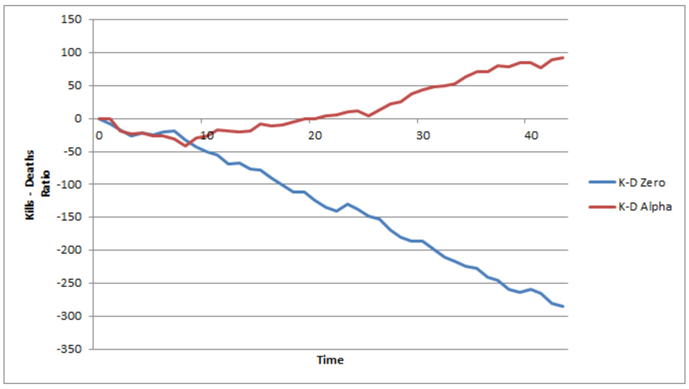
(a) Kills per team.



(b) Deaths per team.



(c) Upgrade points per team.



(d) Kills/Deaths ratio.

Figure 7: Simulation 4

5.5.5 Fifth Simulation

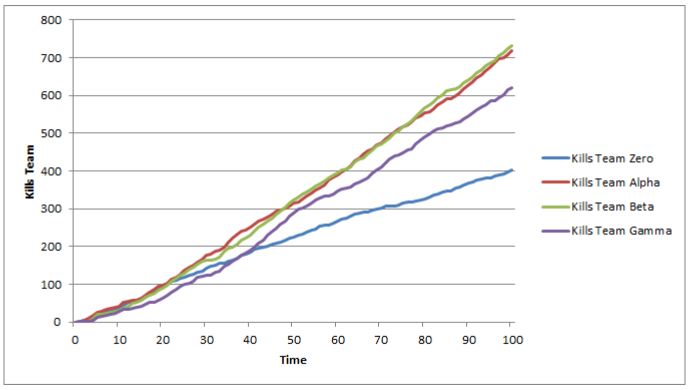
- AI: Goal oriented BT.
- Number of total players: 12.
- Number of players: per team 3.
- Number of teams: 4.
- Number of upgrade points in the level: 25.
- Simulation time: 100 min.
- Simulation system: P-Sense.

Following conclusions were made from the data gathered:

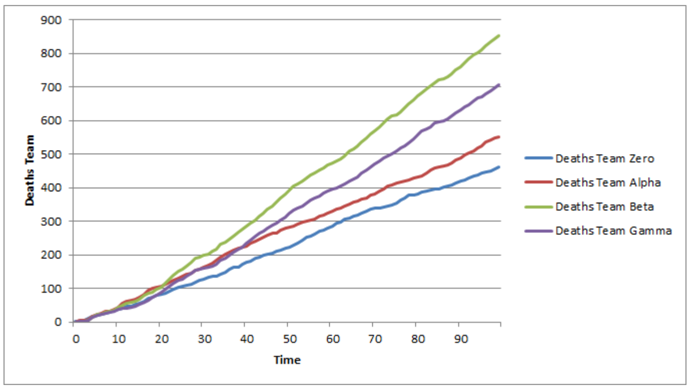
- Team Beta won the match with 13 kills more than team Alpha.
- The kills-deaths ratios were negative the first 20 minutes as shown in **Figure 8(d)**.
- Team Beta played this time the kamikaze strategy. This increased both deaths and kills rate as **Figures 8(a)** and **8(b)** shows.
- The first 40 minutes of the game team Zero got the most upgrade points in the level as shown in **Figure 8(c)**; this advantage decreased the amount of deaths over time as shown in **Figure 8(b)**. But the amount of kills over time was decreased as shown in **Figure 8(a)**. The reason for this behaviour is that team Zero was divided in capturing upgrade points. The teams in this simulation only got 3 players, so if two players decided to capture upgrade points and the last concentrate on killing enemies, this team will only have 33% of fire power. Team Beta on the other hand concentrated all ships in killing enemies.
- Team Alpha got the highest K-D Ratio. This statistic is proportional to the number of upgrade points.

The results for this simulation are very similar to the ones of the first and second simulations. This also means that:

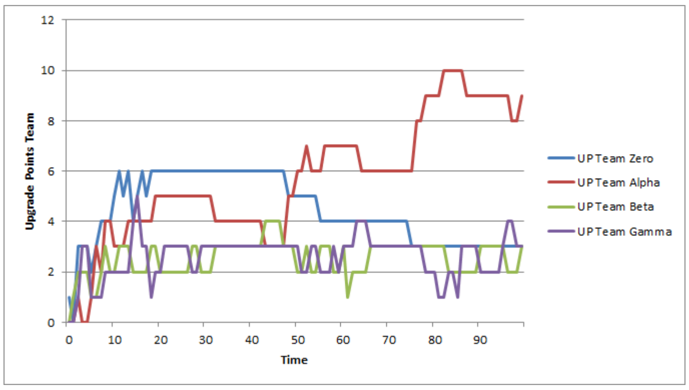
- The number of teams increased the chance of winning the game due kamikaze tactic.
- The winning team using this strategy will always have a negative Kills-Deaths Ratio.



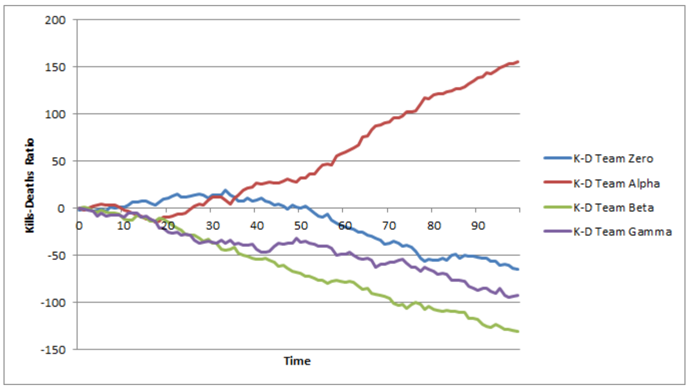
(a) Kills per team.



(b) Deaths per team.



(c) Upgrade points per team.

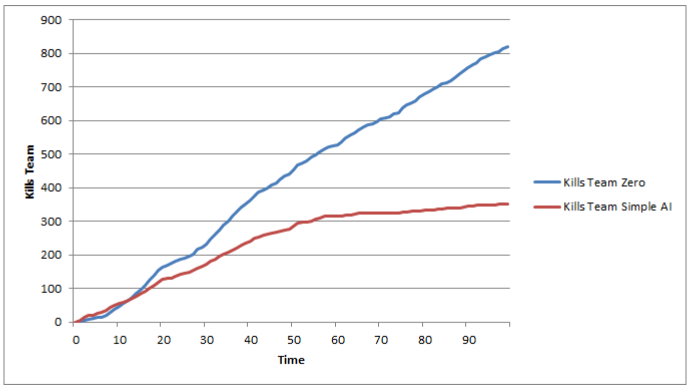


(d) Kills/Deaths ratio.

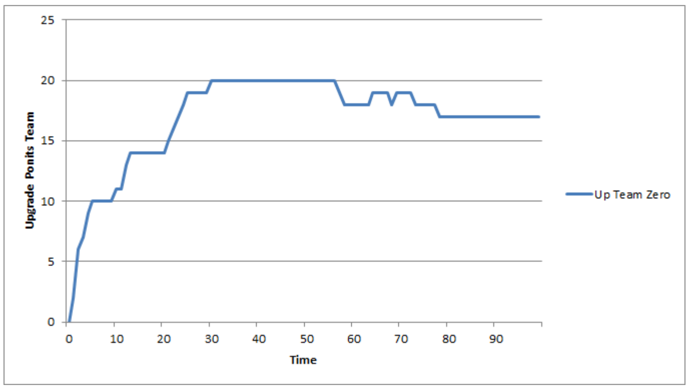
Figure 8: Simulation 5

5.5.6 Sixth Simulation

- AI: Team Zero using Goal oriented BT and Team Simple AI using FSM.
- Number of total players: 10.
- Number of players: per team 5.
- Number of teams: 2.
- Number of upgrade points in the level: 25.
- Simulation time: 100 min.
- Simulation system: P-Sense.



(a) Kills per team.



(b) Upgrade points per team.

Figure 9: Simulation 6

The goal of this simulation was to demonstrate that the goal oriented BT AI obtains better results than the FSM AI. The reason behind this is not the actual AI system, but how they are configured. The goal oriented BT AI is configured to play as a team. On the other hand, the FSM AI was configured only to fight enemies. As shown in **Figure 9(a)**, at the beginning of the game the FSM AI obtained a little of advantage over the BT AI. But after the BT AI got control over some upgrade points the difference was overwhelming.

Simulation scenarios	AI	Players / Team	Teams	Total Players	Upgrade Points / Level	Sim. Time	Sim. System
1	Goal Oriented BT	3	3	9	36	100	P-Sense
2	Goal Oriented BT	4	3	12	36	100	P-Sense
3	Goal Oriented BT	5	2	10	36	100	P-Sense
4	Goal Oriented BT	10	2	20	25	48	P-Sense
5	Goal Oriented BT	3	4	12	25	100	P-Sense
6	Goal Oriented BT / FSM	5	2	10	25	100	P-Sense

Table 4: Summary of all simulations.

5.6 Summary of results

The conclusion from the results obtained from the simulations is that, the behaviour of an AI does not only depend on how smart the AI is, but also on the environment where it is. That is, a single AI strategy does not fit well in all possible environments. The environment is represented by several factors, such as the amount of asteroids in the area, the number of teams, the total number of upgrade points in the area or the number of players per team. Each one of these aspects affects the behaviour of the AI, for instance if the AI is in a place where there are a lot of asteroids, the AI will just concentrate in evading the asteroids rather than fighting the enemy or capturing enemy upgrade points.

The goal oriented BT AI implemented in this thesis has shown to be able to successfully make use of the bonuses given by the game, such as shield generators and upgrade points. These upgrades give an advantage in terms of statistics, such as the Kills-Deaths Ratio, which is the one that benefits the most from the number of upgrade points. It was noted that the Kill-Death Ratio is a good indicator of the likelihood for a team to win a game in all considered scenarios, and especially in games with only two teams. In this case the use of upgrade points is indispensable since the team with the most upgrade points is the winner accord to simulations fourth and fifth.

Another behaviour observed during the simulations was the use of the kamikaze strategy. In this strategy the bots prioritized combat regardless of how many deaths they receive. For scenarios with only two teams this is a bad strategy since dying implies giving kills to the other team. However Planet π4 allows playing with an unlimited number of teams; in a scenario with many teams the kills generated by the kamikaze strategy are distributed among all other teams, does not have a strong negative impact like when playing with two teams only. Indeed in most of the simulations where more than 2 teams were playing, this strategy prevailed over the others. The fact that a strategy like kamikaze can be successful is caused by to the current respawn system, in which players respawn in the middle of the map. Respawning where the enemy is located, usually means to combat immediately after respawning.

At the current stage and based in the simulation results, the implemented goal oriented BT AI achieves the goal of this thesis: the AI plays as a team using the bonuses of the game and reporting the position of the enemies when they are in sight.

6 Conclusions

In this chapter we present a summary of the results and major conclusions achieved during the realization of this thesis. The chapter follows with a scope for future work on the peer-to-peer game Planet π4 and the goal based BT AI implemented in this thesis.

6.1 Summary

In order to accomplish the goal of good game experience and performance without the limitations of a client-server architecture, the implementation of the prototype research MMOG Planet π4 running over a peer-to-peer system is proposed. The focus of this thesis is to create an AI for the game Planet π4. The AI is intended to simulate human behaviour such as work as a team, use the chat system to inform where enemies are and exploit the bonuses that the game offers.

In this thesis we investigated the state of the art of AIs for computer games, from the research results we chose to work with FSM and goal oriented BT AIs. The reason for choosing these two system over HTN and GOAP is that HTN And GOAP uses planners. Planners are great to create complex AI behaviour but that comes at cost of processing time. FSM and BT achieve the goal of this thesis without the necessity of using a planner.

The first implemented AI uses FSM and it was designed in order to create simple combat behaviour. The main objective of this AI is to test how much better is an AI that plays intelligent against an AI that only rushes in combat. The second implemented AI is based on goal oriented BTs and is meant to be smart as a human player using different strategies and tactics to win the game. Both AIs use the same system to evade asteroids. This system is very basic and can be improved in the future.

The game Planet π4 has improved greatly over the last few months, improving the gameplay and the game logic[20]. With the current state of the game, players can not only fight against other players but they can also use the environments as advantage such as hiding behind asteroids, capturing upgrade points or recovering their shield in “shield generators stations”. Along the way, the implementation of the AI had to adapt to these changes and also to improve its way to play the game.

In order to evaluate the performance of the AIs, several simulations were done using the simulation system P-Sense [2]. The simulation results showed that the two different implemented AIs can be used to test Planet π 4 over peer-to-peer with several clients. It also showed that the goal based BT AI is able to use the bonuses offered by the game and it makes good use of the chat system informing the position of enemies. The BT AI tended to use tactics to control most of upgrade points in the level. This tactic was most successful when playing with two teams, because the only one that benefits from this tactic was the team with the most upgrade points. On the other hand, when there are more teams, this tactic was not successful. The team with the greatest amount of upgrade points did indeed receive fewer deaths than the opposing teams, but they also did not have as much kills as the other teams. This means that the upgrade points are key to win the game in matches were only two teams are in the game, but when more teams are in the game, the AI would not receive much benefit in playing smart when the other teams are playing kamikaze, increasing their amount of kills and deaths very fast. One possible solution to this problem is to penalize players for dying in the game.

Another problem noted in the simulations was the respawn system. With the current build players and bots respawn in the middle of the level every time they are killed or entered the game. Taking in count this situation the bots will have the following behaviour:

If the bot respawns and sees that there is an enemy in its vision range, the bot prioritizes the enemy

and attacks the enemy player. When the bot kills the enemy, the enemy respawns immediately not far away where he died. Then the enemy sees the bot and the procedure will repeated several times. A possible solution is for teams to have a safe base like in games like Unreal Tournament III [11], bots and players respawn in one of the bases under their control.
Most of the tests executed in this thesis were done with a relative small number of bots with a maximum of 20 bots. The reason behind this is that the current version of the simulator P-Sense can only run determined number of peers. Simulations with 30 bots were not possible because at some point the simulator starts to simulate very slowly. During the simulations it was not possible to measure precisely the CPU and RAM consumption of each implemented AI, as these metrics are not yet available in the simulator.

Concluding in this work we created an AI that can be adapted to different game environments, showing a smart behaviour to handle different situations. The current version of the AI can be used to support the current research of the prototype research Planet π4.

6.2 Future Work

The game Planet π4 and the AI from this thesis can be improved in several areas.

The game needs a goal. This goal should represent how players can win the game. A suggestion could be for teams to capture as many upgrade points as possible or to have missions for each team, such as destroy a specific team by capturing all the upgrade points under their control. This will make the game livelier and it will improve the gameplay.

Another important change that has to be done in the game is to set the respawn points for each team in an adequate way. A possible solution could be as mentioned in section 6.1, to respawn by the upgrade point that is under control of the team and that is not under attack. This solution makes the kamikaze strategy useless, forcing the bots and the players to use wiser tactics to win the game.

The simulator must be improved in order to simulate several peers. With the current build it was impossible to simulate 100 minutes with 20 bot.

The AI can be improved in different ways. First thing that can be improved is the current method to evade asteroids. At its current state, the bot evades asteroids if it is close enough to them. Making a 3d graph of the map and using A* as a path finder is a good candidate to evade asteroids. A* is used to find the shortest obstacle free path. This solution can improve the evasion of the bots but it increase the processing time for the bots, resulting in longer simulation time.

Communications between bots can be improve by increasing the variety of the messages. Messages that reports where are enemy upgrade points could help the team. Another recomendations to improve this are is that bots give orders to other bots. These orders are send via team chat with the corresponding *PlayerId*. Bots will ignore the message if the *PlayerId* do not match the *PlayerId* of the bot. If it is required to send the same order to several bots, creating groups in the team can help solving this issue.

In general the AI can be improved if the number of goal and goal evaluators increased. Such increment allows the bot to be more flexible in different situations.

References

[1] ArenaNet/NCsoft. Guild wars, 2005.

[2] Michael Stieler Arne Schmieg and Sebastian Jeckel. psense - maintaining a dynamic localized peer-to-peer structure for position based multicast in games.

[3] Ilyas Cicekli Ayse Pinar Saygin and Varol Akman. Turing test: 50 years later.

[4] BioWare/Interplay. Baldur's gate, 1998.

[5] Monolith Productions/Warner Bros. F.e.a.r., 2005.

[6] Monolith Productions/Warner Bros. F.e.a.r. 2: Project origin, 2009.

[7] Mat Buckland. Programming game ai by example.

[8] Robin Baumgarten Chong-U Lim and Simon Colton. Evolving behaviour trees for the commercial game defcon.

[9] Blizzard Entertainment. World of warcraft, 2004.

[10] Valve Software/Sierra Entertainment. Half-life, 1998.

[11] Epic Games. Unreal tournament iii, 2007.

[12] Maxis/EA Games. Spore, 2008.

[13] Double Helix Games/Konami. Silent hill: Homecoming, 2008.

[14] Guerrila/SCEA. Killzone 2, 2009.

[15] Hector Munoz-Avila Hai Hoang, Stephen Lee-Urban. Hierarchical plan representations for encoding strategic game ai, 2005.

[16] http://irrlicht.sourceforge.net/. Irrlicht game engine.

[17] http://www.gamerankings.com/pc/914653-guild wars/index.html. Guild wars review.

[18] Epic Games/GT Interactive. Unreal tournament, 1999.

[19] Chun-Ying Huang Kuan-Ta Chen, Polly Huang and Chin-Laung Lei. Game traffic analysis: An mmorpg perspective, 2005.

[20] Denis Lapiner. Gameplay design and implementation for massively multiplayer online game.

[21] Alejandro Buchmann Max Lehn, Tonio Triebely and Wolfgang Effelsberg. Benchmarking p2p gaming overlays.

[22] Christof Leng Alejandro Buchmann Max Lehn, Tonio Triebely and Wolfgang Effelsberg. Performance evaluation of peer-to-peer gaming overlays.

[23] Jeff Orkin. Three states and a plan: The a.i. of f.e.a.r., 2006.

[24] Abdennour El Rhalibi and Madjid Merabti. Agents-based modeling for a peer-to-peer mmog architecture.

[25] Sega. Virtua fighter 2, 1997.

[26] Bethesda Softworks. Fallout 3, 2008.

[27] Bungie Software/Microsoft Game Studios. Halo: Combat evolved, 2001.

[28] Bungie Software/Microsoft Game Studios. Halo 3, 2007.

[29] www.4gamer.net/specials/gametrics/gametrics.shtml. 4gamer.net. gametrics weekly korea mmorpg population survey.

[30] Billy Yue and Penny de Byl. The state of the art in game ai standardisation, 2006.