Rok akademicki 2010/2011

Politechnika Warszawska

Wydział Elektroniki i Technik Informacyjnych

Instytut Informatyki



praca dyplomowa INżynierska

Piotr Gwizdała

Optimizing intelligent behaviour   
in a First-Person Shooter computer game

Opiekun pracy  
dr inż. Paweł Cichosz

Ocena:

Podpis Przewodniczącego

Komisji Egzaminu Dyplomowego

Specjalność: Inżynieria Systemów Informatycznych

Data urodzenia: 1987.06.02

Data rozpoczęcia studiów: 2006.10.02

Życiorys

Urodziłem się 2 czerwca 1987 roku w Słupsku. Od września 2003 uczyłem się w klasie o profilu matematyczno-fizycznym w III Liceum Ogólnokształcącym im. Marynarki Wojennej w Gdyni, gdzie w lipcu 2006 uzyskałem świadectwo dojrzałości. Od października 2006 studiuję na Wydziale Elektroniki i Technik Informacyjnych Politechniki Warszawskiej. Od września 2009 pracuję jako programista Java.

Podpis studenta

EGZAMIN DYPLOMOWY

Złożył egzamin dyplomowy w dniu 2011 r.

z wynikiem

Ogólny wynik studiów:

Dodatkowe wnioski i uwagi Komisji:

OPTYMALIZACJA INTELIGENTNEGO ZACHOWANIA   
W GRZE PERSPEKTYWY PIERWSZOOSOBOWEJ

Praca ta przedstawia autonomicznego agenta zaprojektowanego na potrzeby komercyjnej gry perspektywy pierwszoosobowej. Parametry stworzonego agenta zostały zoptymalizowane w celu uzyskania możliwie efektywnego zachowania.

Powiązania między dziedziną sztucznej inteligencji a komercyjnymi grami komputerowymi zostały omówione. Opisano metody sztucznej inteligencji wykorzystywane w grach perspektywy pierwszoosobowej oraz podstawowe metody optymalizacji stochastycznej. Stworzony agent, wykorzystujący logikę rozmytą do podejmowania decyzji, został szczegółowo opisany.

Parametry algorytmów i procedura optymalizacji zostały odpowiednio dobrane i uzasadnione. Zastosowano cztery algorytmy optymalizacji stochastycznej i porównano ich wyniki. Ostateczny rezultat został porównany z zewnętrznym agentem i z graczami ludzkimi, by ocenić skuteczność optymalizacji i konkurencyjność znalezionego rozwiązania.

Słowa kluczowe: sztuczna inteligencja, gry komputerowe, optymalizacja stochastyczna

OPTIMIZING INTELLIGENT BEHAVIOUR   
IN A FIRST-PERSON SHOOTER COMPUTER GAME

This thesis presents an autonomous agent designed to play a commercial First-Person Shooter computer game. Agent’s parameters have been optimized to behave in a possibly effective way.

Connections between artificial intelligence and commercial computer games have been outlined. Artificial intelligence methods used in First-Person Shooter games and basic stochastic optimization algorithms have been described. The agent developed using a fuzzy logic approach to decision-making has been described in detail.

Optimization algorithms parameters as well as the optimization procedure were chosen and justified. Four stochastic optimization algorithms were applied and the results were compared. The final result has been evaluated against a popular third-party game agent and human players to assess the effectiveness of the optimization and the competitiveness of the solution found.

Keywords: artificial intelligence, computer games, stochastic optimization

Contents

[1 Introduction 1](#_Toc284070872)

[1.1 Artificial Intelligence 1](#_Toc284070873)

[1.2 Computer Games and AI 2](#_Toc284070874)

[1.3 AI methods in computer games 3](#_Toc284070875)

[1.4 First Person Shooter games 4](#_Toc284070876)

[1.5 Motivation 4](#_Toc284070877)

[1.6 Thesis Overview 5](#_Toc284070878)

[2 Background 7](#_Toc284070879)

[2.1 Artificial Intelligence in First-Person Shooter games 7](#_Toc284070880)

[2.1.1 Bots architecture 7](#_Toc284070881)

[2.1.2 Navigation solutions 9](#_Toc284070882)

[2.1.3 Finite State Machines 12](#_Toc284070883)

[2.1.4 Fuzzy Logic 13](#_Toc284070884)

[2.2 Optimization methods 16](#_Toc284070885)

[2.2.1 Definition 16](#_Toc284070886)

[2.2.2 Stochastic optimization 18](#_Toc284070887)

[2.2.3 Random search 19](#_Toc284070888)

[2.2.4 Hill-climbing 22](#_Toc284070889)

[2.2.5 Finite-difference stochastic approximation 24](#_Toc284070890)

[2.2.6 Simulated annealing 27](#_Toc284070891)

[2.3 Quake II 30](#_Toc284070892)

[2.3.1 Quake II deathmatch 30](#_Toc284070893)

[2.3.2 Quake II client-server communication 31](#_Toc284070894)

[2.4 QASE API 32](#_Toc284070895)

[2.5 EraserBot 33](#_Toc284070896)

[3 Developed solutions 35](#_Toc284070897)

[3.1 Creating a bot using QASE API 35](#_Toc284070898)

[3.2 Knowledge representation 35](#_Toc284070899)

[3.3 The main concept 37](#_Toc284070900)

[3.4 Navigation plan 38](#_Toc284070901)

[3.4.1 Criteria for choosing destination item 39](#_Toc284070902)

[3.4.2 Fuzzy logic application 42](#_Toc284070903)

[3.4.3 Enemy engaging plan 44](#_Toc284070904)

[3.5 Firing decision and instructions 45](#_Toc284070905)

[3.5.1 Weapon and enemy choice 45](#_Toc284070906)

[3.5.2 Aiming algorithm 45](#_Toc284070907)

[3.6 ReferenceBot and LearnBot 47](#_Toc284070908)

[4 Experiments 49](#_Toc284070909)

[4.1 Initial experiment 49](#_Toc284070910)

[4.1.1 Experiment description 49](#_Toc284070911)

[4.1.2 Results 50](#_Toc284070912)

[4.2 Optimization 51](#_Toc284070913)

[4.2.1 Results measurement 53](#_Toc284070914)

[4.2.2 Choosing optimization parameters 55](#_Toc284070915)

[4.2.3 Localized random search 56](#_Toc284070916)

[4.2.4 Hill-climbing 57](#_Toc284070917)

[4.2.5 Finite difference stochastic approximation 59](#_Toc284070918)

[4.2.6 Simulated annealing 60](#_Toc284070919)

[4.2.7 Algorithms comparison 61](#_Toc284070920)

[4.3 Evaluation 63](#_Toc284070921)

[4.3.1 Human player study 66](#_Toc284070922)

[4.4 Summary 68](#_Toc284070923)

[5 Conclusions 69](#_Toc284070924)

[5.1 Main contributions 70](#_Toc284070925)

[5.2 Further work 71](#_Toc284070926)

[6 Attachments 73](#_Toc284070927)

[6.1 Glossary 73](#_Toc284070928)

[7 Bibliography 75](#_Toc284070929)

# Introduction

In the modern Western World, the creation, distribution and manipulation of the information has become our main cultural and economical activity. We are transforming ourselves into *information society*. Many believe that this particular paradigm shift may have a similar impact on human lives as the Neolithic Revolution, when we transformed from the hunting and gathering to the agriculture-based way of living. Vivid visions of the future of the information society invariably present our lives filled with computers, which are often more intelligent than most of humans. Nevertheless, in close future, the superhuman or even human-like intelligence, is not very likely to be created – the development of so-called *strong artificial intelligence,* or *strong AI* remains mainly a domain of a theoretical debate [1].

At the same time, the *weak AI,* that focuses rather on particular problems that require intelligence to be solved, advances rapidly and it has currently become an inseparable part of our lives – the methods of artificial intelligence (AI) are used in weather forecasts, digital photo cameras, internet search engines, business planning, medicine, and many others. One of the examples of a predominant field of its application, and at the same time an appropriate research platform, are commercial computer games [2], which this thesis will focus on.

The first part of this chapter covers the core concepts that have been used in this thesis – namely, the artificial intelligence, the computer games and their relation with AI, AI methods used in games, as well as the First-Person Shooter games genre. Further on, the motivation that stands behind the idea of using computer games for an AI research will be presented. Eventually, the chapter ends with a brief description of the thesis goal and the outline of its content.

## Artificial Intelligence

The term *artificial intelligence* is closely related to the concept of human intelligence, which can be defined as “a mental quality that consists of the abilities to learn from experience, adapt to new situations, understand and handle abstract concepts, and use knowledge to manipulate one’s environment” [3].

The main goal of the artificial intelligence is to create a machine, that would be intelligent. Nonetheless, there are various approaches and ideas on what the artificial intelligence should be like and how it should be achieved. Some believe that AI should be developed following the example of the human brain, seeing AI as an empirical science, that involves hypothesis and experimental confirmation. On the other hand, others understand AI as a bottom-up development of a rational system using a combination of mathematics and engineering. To arrange these different approaches, Russel and Norvig [4] propose the following classification of AI systems:

* Acting humanly – systems that can pass the Turing Test – a test proposed by Alan Turing [5] that is considered to be a good operational definition of intelligence. In the test, the judge would communicate using text-based interface with the subject without knowing whether it is another human or the machine. If the judge incorrectly recognizes the machine as a human, the machine is said to have passed the Turing Test.
* Thinking humanly – systems that have been developed basing on the knowledge of how human mind works – which is the main concern of the field of cognitive science. Once we understand how human brain works, providing that we will have sufficiently powerful computer, we may be able to simulate the human brain on it.
* Thinking rationally – systems that are reasoning using the means of formal logic. One of the main limitations here is that in order to interact with the real-world environment, the system needs to know how to transform the informal, often uncertain knowledge, and subsequently state it with the use of formal terms required by its logical notation.
* Acting rationally – rational agents systems, i.e. agents that act in the best possible way in order to achieve their goal. Even if the agent has only uncertain or incomplete information, it should still make a decision, even if it is an not optimal one.

Currently in the field of AI, only the *weak AI* systems have been realised – systems that are able to solve just some specific set of problems. Usually, they fit in one of last two categories – systems that think or act rationally. The AI system that is deliberated throughout this thesis fits best into the fourth category – the rational agents systems.

## Computer Games and AI

The term computer games refers to interactive games operated by computer circuitry [6]. It is believed that the first electronic game was created by William A. Higinbotham – of the Brookhaven National Laboratory, in 1958, as the interactive technology demonstration available for laboratory’s visitors [7]. Tennis for Two worked on analogue computer and simulated a tennis game on an oscilloscope. It demonstrated a simplified projection of a tennis court from the side, featuring a gravity-controlled ball that needed to be played over the net. The players used analogue controllers to adjust the trajectory of the ball and a button to hit it with an invisible racket [7].

Since then, computer games evolved from small programs or devices developed by individuals to major commercial projects produced by teams of experienced developers working, in some cases for years, on a particular product. The whole industry is estimated to be worth $11.7 billion in 2008 just in United States, which places it in front of music and film industries [8].

In most computer games, the players interact not only with other human players, but also with non-player characters (NPCs) – characters appearing in the game that are controlled by game program and not by a human player. In order to provide more realistic, human-like behaviour of NPCs, game creators started to use some of the techniques developed in the field of artificial intelligence (AI). The term game AI appeared relating to NPCs controllers simulating intelligent behaviour in a computer game. Challenging and entertaining game AI has been broadly recognized as a second most important factor in a particular game’s commercial success, with only graphics being more important [9].

## AI methods in computer games

Almost every computer game AI has to navigate through the game world. The task of navigation is to decide where the agent should move, whilst the task of path-finding is to decide which way the agent should go in order to reach its destination. Often, the abstract representation of the game world is used, so the agent can perform the path finding more efficiently. Most of computer games use a simple finite-state machines (Subsection 2.1.3) to make the navigation decision. However, approaches using a fuzzy logic (Subsection 2.1.4) were shown to give better effects, like in *The Sims*, which was widely acknowledged for the depth of personality of its AI agents [9].

When agent’s parameters and configuration need to be tuned in order to reach the level of performance close to human players, the optimization algorithms (Section 2.2) are often applied, like in case of [10]. However, the time-consuming optimization is usually performed during the development process of the game.

Agents that optimize or adapt their behaviour during the game are still not very common. One of the reasons is that the game environments are often very complex and stochastic, setting a great challenge for the domain of machine learning. Some commercial computer games try to give a false impression of learning by degrading an AI that performs well through addition of random errors. The effect of learning can be achieved by gradually reducing the frequency and magnitude of the errors [11].

## First Person Shooter games

Computer games are divided into relatively small set of different game genres e.g. strategy, sports, racing, adventure etc. However, in many cases it is not possible to associate a particular game with just one genre. One of the most popular and financially significant computer game genres is a First Person Shooter genre, commonly abbreviated with FPS.

In FPS, human players use a mouse and a keyboard, or other input devices, to control their virtual in-game character. The main input for a player is a first-person perspective view of the world displayed on the screen and sounds played in the game. The player sees the view from the eyes of the character he or she controls. The usual scenario in an FPS game focuses on fighting against opponents using some sort of firearms. The player’s character is placed in the three dimensional world together with other opponents, which can be controlled by other human players or by computer programs called bots[[1]](#footnote-2).

All participants of the game can move around the world and pick up weapons and special items such as medical kits and armour jackets. Each FPS game is different, but usually, player’s health is described with some number and, if player’s health is low, it can be recovered with a medical kit. If a player wears an armour jacket, the damage taken from gunshots will be reduced.

## Motivation

Commercial computer games are starting to be perceived as attractive platforms for AI research. One of the reasons is that computer games usually provide a complex, large scale simulation of a real-world environment with realistic physics and vast interaction possibilities. However, in contrary to the real world, the sensing and actuating is simplified and the experiments can be easily controlled and repeated. This allows us to focus on our research task, and hopefully, achieve better results. Commercial computer games are also considered to provide more objective test environment than those developed on purpose for a given research [2].

The FPS genre is particularly considered to be attractive for research, as player’s actions usually have direct influence on the state of a game and his environment [12]. In a football simulation game, for instance, the team’s result depends not only on a particular player’s actions, but also on actions of other team members. Secondly, there are many similarities between controlling a character in FPS game and controlling a real-world mobile robot, like a problem of path-finding.

The attention given to commercial computer games is increasing within the AI research community. For instance, in order to stimulate research, Philip Hingston [13] proposes a variant of Turing Test [5] designed for FPS bots on which the BotPrize competition is based. In the competition, taking place every year since 2008, human players play with bots an FPS game, while being observed by judges. Basing only on observed game character’s behaviour, the judges have to tell the human players from the bots. Until now, none of the bots have managed to appear human-like enough to win the BotPrize [14].

## Thesis Overview

There are two main objectives of this thesis:

1. Design and develop an autonomous FPS game agent, using a fuzzy logic approach for decision-making, that is able to compete with other players.
2. Improve the developed solution with a use of optimization methods and compare it with third-party bots and human players.

The first objective will allow us to examine what the difficulties of creating an autonomous agent in a complex environment of the FPS game are, and to test in practice the methods of the game AI, providing us with a base for further development.

The completion of the second objective will give us more profound understanding of the problem of optimization in a stochastic FPS game environment, and will let us evaluate the suitability of the optimization methods used in this application.

While completing these objectives, we will try to answer the main research questions:

* *What is the effect of different optimization algorithms on the performance of the FPS game bot?*
* *Is optimization an effective way to achieve a more intelligent behaviour of the FPS game bot?*

In the following chapter the methods of the AI used in FPS games have been outlined, followed by the description of the basic optimization algorithms along with their stochastic equivalents. Later, Quake II FPS game is introduced and QASE API – a framework that facilitates AI-related research. Chapter 2 ends with a brief description of EraserBot – a third-party Quake II agent, that will be used in the experiments described in Chapter 4.

Chapter 3 provides a description of the solution developed in this thesis, starting with a general idea and outline of the algorithm of the agent. Next, the navigation module using fuzzy logic is explained in detail, as it’s parameters will be optimized in Chapter 4. The chapter ends with the description of bot’s combat module – the aiming algorithm and the choice of the enemy and the weapon.

Chapter 4 describes and presents the results of the experiments that have been performed. The parameters are chosen and the optimization results are presented for four different stochastic optimization algorithms. The results are compared and the best configuration found is evaluated against the third-party agent. The chapter ends with a short human player study.

Finally, the Chapter 5 draws the conclusions, summarizes the main contributions of this thesis and discusses some possible future work.

# Background

The first chapter presents the core concepts of AI and computer games, and reveals the benefits of using the computer games in AI research. Furthermore, the chapter introduces the FPS genre and argues the reason of its attractiveness from the AI researchers’ point of view.

In the beginning of this chapter, the basic methods used by developers of the AI in FPS games are presented. Subsequently, some optimization algorithms along with their stochastic equivalents are introduced. In the next part, the Quake II is presented as an example of the FPS game, followed by description of QASE API – a framework for facilitating the high-end AI-related research. The chapter ends with a brief description of EraserBot – an example of Quake II bot.

## Artificial Intelligence in First-Person Shooter games

In this section the basic concepts and methods often used while developing the FPS game AI are described.

### Bots architecture

The form of the FPS game determines a set of basic actions that all the players need to perform. This includes navigating through the three dimensional world, selecting an appropriate item or a gun to use, aiming and shooting at the enemies.

The set of basic actions, that a player needs to perform in FPS game can be a good starting point to develop a generic architecture of FPS AI. Paul Tozour [15] proposes an architecture divided into four main components: animation, movement, combat and behaviour. Figure 1 presents a diagram of those four basic components.

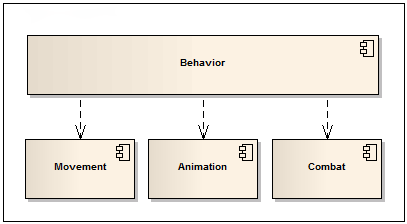


Figure 1: UML component diagram of an FPS game AI architecture proposed by Tozour [15].   
A dashed arrow represents a functional dependency between components.

The animation component is responsible for controlling the character’s virtual body. This can be done by adjusting parameters of existing animations (e.g. character’s running speed), playing a right animation at a right time (e.g. climbing up the ladder) or by solving an inverse kinematics problem[[2]](#footnote-3), when a character reaches for an item. This component should also control which parts of the body perform which animation and deal with conflicts (e.g. bot death animation should have higher priority than bot jump animation).

Bot’s movement or navigation controller provides a service for other components – it allows them to move a bot from its current position to a specified one. This task requires a bot to perform path finding. It has to decide following which path it will move towards its destination. The path is usually represented as a sequence of points in the world, that a bot has to follow, which involves using some abstract representation of the game world, called a map. In the next subsection closer insight into common game map representations will be provided. After the path has been established, the movement controller turns the character in the right direction and controls its movement from one point of the path to another. Also, if some dynamic obstacles appear, the movement controller should respond appropriately – trying to solve the problem or reporting it.

When a bot enters the combat, the combat controller should take the control over most of bot’s behaviours, such as a weapon and an opponent selection, firing and manoeuvring or picking up items. The main challenge here is to evaluate a situation quickly and choose an appropriate tactic, which appears to be quite easy for humans and difficult for computers. One reason for that may be that we are very good at evaluating the spatial configuration of entities in the world, which allows us to make better decisions. For instance, humans quickly find good places to hide from a gunfire or to shoot at the enemy. Modern bots still find this task difficult and base on scripted behaviour, pre-defined by their authors. Another aspect of combat that ought to be controlled by the combat component, is the group tactics and communication between group members during the combat.

The behaviour component is one that controls all the other components and makes high-level decisions about bot’s behaviour. It decides whether the bot should search for an enemy or a better weapon; whether it should enter into combat or retreat. The quality of this component will determine bot’s resulting behaviour.

### Navigation solutions

Spatial reasoning cannot be performed on the raw geometry of the game world. The main reason is complexity. A single brick in a wall can be described with as many as thousands of polygons, with a wall consisting of hundreds of bricks. The only aspect a bot needs to know is the fact that there is a wall. All additional information is not important when performing a path finding. It would make the task computationally expensive, while the bot needs to operate in a real-time. More abstract representation of the game world is necessary.

Regardless the chosen representation, most modern computer games have it prepared by their creators (like in [10] and [16]) before the game is released. Although works like [17] try to make this process automatic, bots are not yet able to learn about the world by themselves, at least not well and fast enough to satisfy game developers’ requirements.

Waypoint map

One of the most popular abstract game world representation is a waypoint map. Generally speaking, a waypoint map is a graph in which nodes represent reachable points in the game world, and the edges indicate that it is possible to move from one node to another (Figure 2).

Edges can be labelled with a distance or with an action necessary to take in order to move from one node to another (e.g. jump or crouch). The nodes, on the other hand, can also contain some additional information, like an item type that can be expected in a given place or that the given node is a good place to hide at.

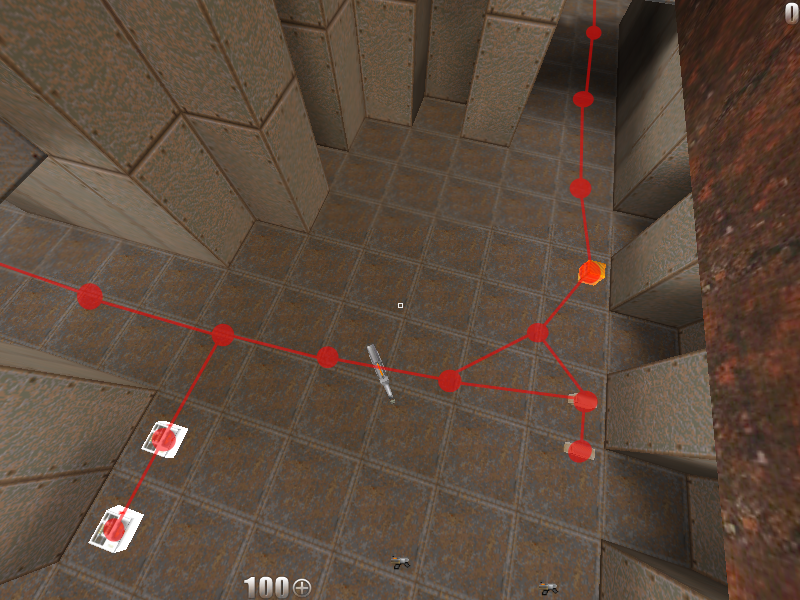


Figure 2: The example of the waypoint map.

It is important to make sure that moving from one node to another along an edge can be easily performed by a navigation module. Usually it means, that the only thing the bot needs to do is to turn towards a destination waypoint and move forward until it arrives there.

Having such a representation of the game world, we can easily navigate between any points on the waypoint map if an appropriate path exists. To perform path finding a graph search algorithms can be used, or if the game environment is static enough, all the paths can be computed before the game. However, it is important to make sure that the path finding works fast enough for a real-time game.

Navigation mesh

In recent years, the navigation mesh has become the world space representation of choice for agents in virtual worlds [17]. It divides all the walkable surfaces of the environment into convex polygons, creating something that can be called a “floor plan” of the world (Figure 3). Navigation mesh can be also represented as a graph in which nodes are polygons, and the graph edge exists between two nodes if their polygons sides overlap.

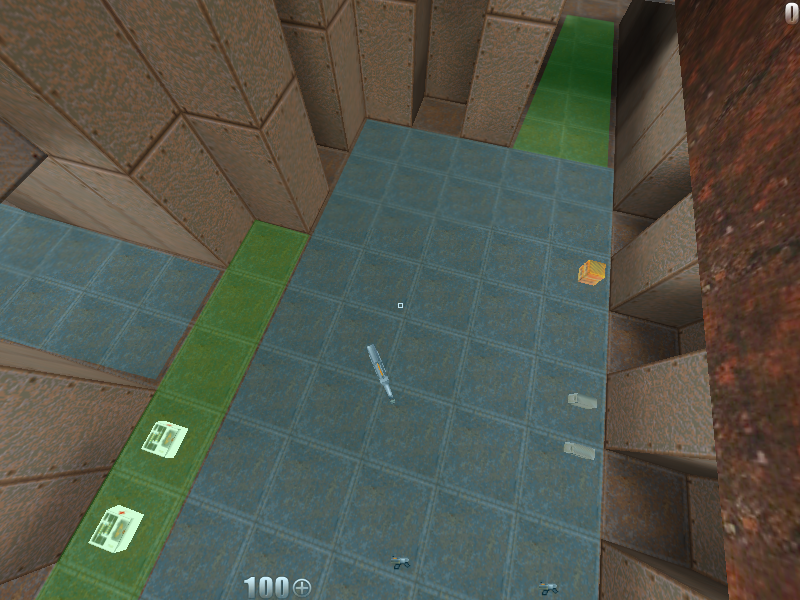


Figure 3: The example of the navigation mesh.

Navigation meshes are considered to be more powerful and providing more realistic navigation [18]. In a waypoint map, a bot could be located only at the waypoints or somewhere on the edge between them. In navigation mesh a bot can walk over the whole surface of each polygon. This allows more flexible, less schematic and more realistic movement, while still being relatively simple representation of the game map (Figure 4). Since we still use a graph, the path finding can be performed in exactly the same way as in case of waypoint maps.

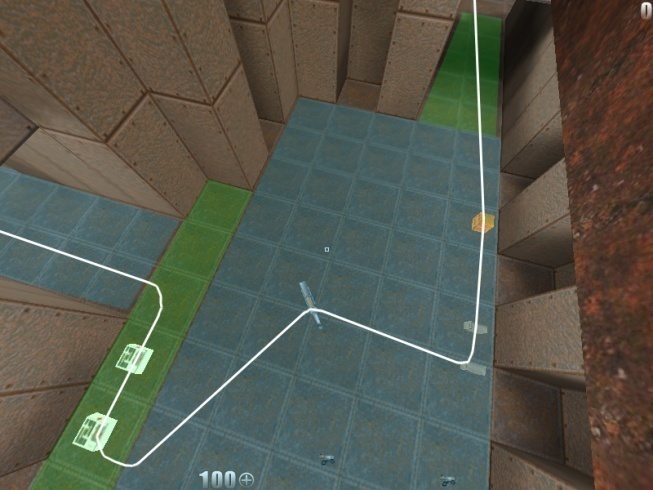
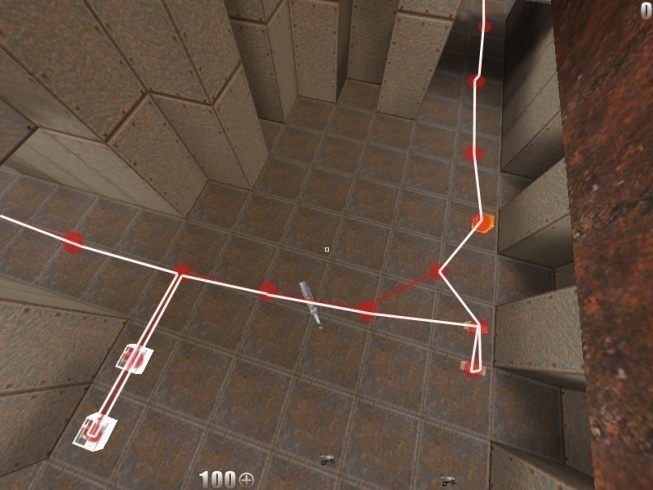


Figure 4: The comparison between the waypoint map and the navigation mesh. On the left picture the white line represents the path that the bot would follow in order to collect all the items in the room. On the right picture the bot performs the same action using the navigation mesh. The navigation mesh allows the agent to move in a more natural way.

However, moving from one polygon to another may require not only finding an appropriate polygon’s edge and moving towards it, but also avoiding dynamic obstacles that may appear on the way. In case of a non-player characters that do not live long enough in the game, being usually shot by a player, it may not be cost-efficient to develop a navigation mesh based movement component. However, if a human player will have enough time to take a closer look at our bot, the navigation mesh can often give a more realistic result.

### Finite State Machines

State machines, along with scripting are two most common techniques used in modern games to perform decision-making. Their popularity can be attributed to their simplicity and their power of expression.

Often game characters will behave in a certain way until some event occurs. For instance, a bot will search for a weapon, but as soon as it sees an enemy it should change its behaviour and decide whether to fight or retreat. This kind of behaviour can be achieved with finite state machines (FSM).

An FSM is a system that has a limited number of states. At a moment only one state is occupied. Each state can be associated with some specific bot’s action. Transitions exist between states. Each transition has a set of associated conditions. If conditions of a transition are met, the machine moves from one state to another. An example of game FSM is presented in Figure 5. In this example, for instance, if a bot is in the state “Search for enemies and fight” and it gets wounded during a fight, the transition to the state “Search for a medical kit” takes place and the bot starts to search for a medical kit in order to heal itself.

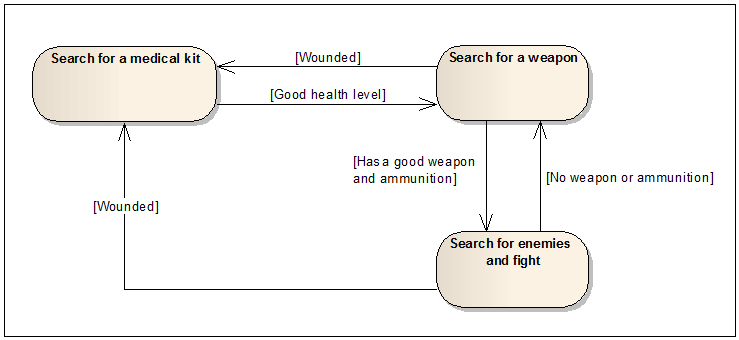


Figure 5: An example of a simple game FSM in UML notation.

Finite state machines are easy to use and read. A designer can also, without too much effort, adjust the level of detail in which the model reflects a desired behaviour by adding or removing additional states and transitions.

### Fuzzy Logic

In a narrow sense, fuzzy logic is a logical system – a generalization of conventional logic [19]. While in conventional logic a variable can either be true or false, in fuzzy logic it can have associated any real number in a range . The 0 value is interpreted as false and 1 as true. The numbers in between express the “degree of truth”.

Fuzzy logic in a wide sense is a term that refers to a union of fuzzy logical system, fuzzy set theory, possibility theory, calculus of fuzzy if-then rules, fuzzy arithmetic, fuzzy quantifiers and all other theories derived from the concept of fuzzy logical systems [19]. Most of these theories provide tools that can be used when taking decisions with estimated values under incomplete or uncertain information.

There have been many successful applications of fuzzy logic in signal processing, pattern recognition, business forecasting, speech processing, robotics control, natural language understanding etc. Computer games are not an exception – fuzzy logic has been used in many commercial games. It is attractive because of calculation speed and ability to model complex behaviour [19] [20]. It is also often used in combination with other techniques, like presented in [21] FuSM – Fuzzy State Machine.

For an FPS bot, fuzzy sets and relations can be useful to express how much a bot wants to do or have something. In the following paragraphs, some basic theory behind fuzzy relations is recalled.

Definition : Fuzzy set

Let be a nonempty set. A fuzzy set in is characterized by a function:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

is called a membership function which expresses the degree of membership of the element in a fuzzy set . A value of 1 is an equivalent of the classical truth, whilst 0 is false. We can note that a fuzzy set is fully determined by the set of pairs:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

The fuzzy sets theory is a generalization of a classical sets theory. If we were to define a classical set using a fuzzy set theory, our membership function could be defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Fuzzy relations

Let’s consider a classical relation. Let be classical nonempty sets. An -ary relation is a subset of the Cartesian product of nonempty sets:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Since a relation is a set, we can use a membership function to define a fuzzy relation. For a classical -ary relation a membership function is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a vector of arguments of the membership function – the elements of sets .

In case of a fuzzy -ary relation a membership function can take all values from the range.

Fuzzy operators

In order to apply fuzzy logic effectively, the logical operators need to be defined. There are many different fuzzy operators sets, but most of them contain basic operators that are similar to conventional logic: complement , intersection and union .

The way these operations are performed may be different depending on each operators set, although the membership function for a complement operator is usually the same:

Definition : Complement operator

Let be a fuzzy relation. The membership function of the complement of the relation is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Following are definitions of intersection and union operators originally proposed by the author of fuzzy sets theory, Lotfi Zadeh:

Definition : Zadeh’s intersection and union operators

Let and be membership functions of fuzzy relations R and S respectively. The membership function of Zadeh’s intersection of and is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

The membership function of Zadeh’s union of and is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Example 2.

To illustrate operations on fuzzy relations let’s suppose we have bots set and items set . Let the fuzzy relation be defined as “bot needs item” and the fuzzy relation be defined as “bot is close to item”. Both relations are defined on Cartesian product .

If we want to choose an item that is close to a particular bot and the bot needs it, we will choose an item that has the highest fuzzy membership value for an intersection of relations and .

Zadeh’s intersection operator always chooses the minimal fuzzy value. Therefore, it depends only on one – the smallest of the input fuzzy values. In our example, for bot and items and we could have: , and for item : and , the outcome of the intersection operation for both items and would be the same and equal , although the “bot needs item” relation for item has significantly higher membership value, which in this example may be important. Analogical problem arises for Zadeh’s union operator.

One of alternatives to Zadeh’s union and intersection operations that do not suffer from the problem described in Example 2.1 is Ron Yager’s operators set [22].

Definition : Yager’s intersection and union operators

Let and be membership functions of binary relations R and S respectively. The membership function of Yager’s intersection of and is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

The membership function of Yager’s union of and is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

For both expressions is a parameter greater than .

These are just two examples of simple, but yet very popular basic operators sets that can be applied on fuzzy sets and relations [19]. In this thesis Yager’s operators will be preferred over Zadeh’s operators for the precise reason mentioned in Example 2.1.

## Optimization methods

When making decisions in a great variety of fields, such as management, engineering, science, medicine or business, the optimization methods are used to obtain the best result under given circumstances. The goal of vast majority of such decisions is to maximize the benefit or minimize the effort required to make it [23]. In this section the principles of deterministic and stochastic optimization will be introduced. Furthermore, basic optimization methods will be described along with their stochastic versions.

### Definition

In mathematical terms, the maximization problem can be described as follows:

Definition : Maximization problem

The maximization problem is defined as follows: Finding the values of a vector that maximize a scalar-valued gain function :

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

The vector is a collection of parameters we need to adjust in the best possible way, reflects allowable values that can take, and function is a scalar measure that summarizes the performance of the optimized system for given values of .

Any minimization problem can be trivially converted to maximization problem by altering the sign of the gain function .

One of the main distinctions in optimization is between local and global optimization. A global maximum is such that , whilst the local maximum is better than any in its vicinity, but is not necessarily a global maximum. In many practical problems we can be satisfied with the reasonably good local maximum and the search for global maximum may be too expensive.

Another important distinction is the nature of the problem, i.e., whether the domain of elements of is continues or discrete. There can also exist hybrid problems in which some of elements of are discrete and other are continues.

The difficulty of the optimization problem is closely related to the dimensionality of the vector . The size of the search space grows exponentially with . For instance, if , and each of the elements of can take 10 values, there are possible values of . Often, each query of the gain function is computationally expensive, which in the case of makes an optimization difficult, and usually requires some additional problem knowledge in order to solve it. This phenomenon is called the “curse of dimensionality” – a term that was coined by Bellman [24].

### Stochastic optimization

When the gain function is deterministic, i.e., for a given the same value of is always obtained, one can use one of many deterministic optimization algorithms that are sometimes even proven to find the globally optimal solution .

However, in real-life problems we often only have a noisy gain measure. This is where stochastic optimization methods may be very helpful. The stochastic optimization problem can be defined as follows [25]:

Definition : Stochastic optimization

Stochastic optimization is an optimization process that has at least one of the following properties:

1. there is random noise in the measurement of ,
2. there is a random choice made in the search direction as an algorithm advances towards the optimal solution.

Some stochastic search algorithms that have the latter property may also be successfully applied when the gain function is deterministic in order to speed-up the desired solution search process or increase the chances of finding a global optimum.

Problems that have the former property can be presented as optimization where the only information we have about the is a noisy defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is random noise, that can depend on currently evaluated vector .

There are fundamental limitations of the optimization with noisy information about the gain function, as in (2.12). The statistical error of the gain function that will propagate to the output of the optimization algorithm can be reduced by significantly increasing the cost in the number of gain function evaluations and averaging the result. However, in a simple case of an independent noise, the error decreases with a rate of , where is a number of function evaluations at each step [25]. Figure 6 illustrates the relation between the number of function measurement repetitions and the error obtained with a simple experiment. We can see, for example, that if we double the from to , the error will drop, relatively, just by .

test.emf

Figure 6: A mean noisy input measurement error in the function of measurements repetitions obtained with a simple experiment. The error decreases with a rate of , where is a constant. In consequence, for higher values of , a significant increase of , causes only a small decline of the error.

This cost grows even more if the noise is somehow dependent on the evaluated vector , as even more gain function evaluations are usually required.

In the following subsections the stochastic optimization algorithms used in this thesis are outlined.

### Random search

In general, direct random search methods base on searching for the best solution by exploring the domain in a random manner. These are the simplest methods that can be applied in the case of stochastic optimization problem. Although simple, random search in problems proves to be quite effective. Apart from that, these methods are often preferred over other algorithms because of the following reasons [25]:

* Random search is easy to implement – therefore the engineers can come up with the solution faster.
* Using exclusively the gain function – there are no additional requirements to the function, the gradient or hessian are not required.
* Reasonable efficiency – random search algorithms are often able to provide a reasonable solution quite quickly.
* Generality –can be applied to virtually any problem having only the gain function.
* Theoretical foundations – supporting theory exists that guarantees the results and provides means to estimate the expected accuracy of the solution, whilst many of popular optimization algorithms are not theoretically proven to converge to an optimal solution.

In its simplest form, the direct random search algorithm in its -th iteration would simply generate a random vector , following chosen probability distribution, and compare it with the best currently known solution using the gain function. For the step , the better of and will be chosen, following the concept of a greedy algorithm.

The direct random search algorithm samples from the whole domain in the search for the best solution, it does not take into account the current best known solution. On the other hand, the following algorithm does use the best known current solution . This property is sometimes referred to as localized search. This term should not be confused with the local and global optimizations which were mentioned in Subsection 2.2.1. In fact, under certain conditions, some of the algorithms that have a property of localized search, are guaranteed to find the global solution.

Algorithm : Localized random search

1. Randomly choose initial solution . Calculate .
2. Randomly choose an independent vector , where is the dimensionality of . Let , make sure that .
3. Calculate . If then let , otherwise .
4. Stop if the maximum number of evaluations has been reached, or is sufficient. Otherwise, go to step 2, increasing .

In order to apply the localized random search, we need to choose the probability distribution using which vector will be generated. The mean of the distribution should be zero and the standard deviation for each element of the vector should be consistent with its possible values range. For instance, if the first element’s value is in the range of , and the range of values of the second element is , then the variance for the second parameter should be about 10 times greater than for the first one, allowing the desired exploration for each of vector elements. It is often beneficial to decrease the variance of , as the algorithm advances.

The main difference between localized and direct random search is the way the new random solution is generated. In direct random search, the new solution is set with independently generated, new random vector. On the other hand, in the localized random search the new solution depends on the current best known solution . If the variance of the distribution used to generate is relatively small, the algorithm will search locations close to most of the time.

Another possible improvement of the Algorithm 1 is based on observation that if the random vector is added to currently best solution, it decreases the value of the gain function; then adding is likely to result in the increase of the gain function value.

Noisy gain functions

The Algorithm 1 uses the perfect, noise-free measurement of the gain function. If we optimize the noisy gain function instead, the algorithm will generally not converge. However, there are means by which the influence of the random noise can be reduced.

One possibility is to perform measurements of and use the average of the results as if it was a perfect measurement of . However, as mentioned in Subsection 2.2.2 and illustrated in Figure 6, this approach requires many function evaluations in order to decrease the noise by little.

Another approach is to modify the algorithm in a way that the new solution will be accepted only if it there is a significant probabilistic evidence that it is in fact better than the current solution. This can be achieved with the modified version of the Algorithm 1, where the third step is changed as below:

Algorithm : Noisy gain function localized random search

1. Randomly choose an initial solution . Calculate .
2. Randomly choose an independent vector , where is the dimensionality of . Let , make sure that .
3. **Calculate . If then let , otherwise .**
4. Stop if the maximum number of evaluations has been reached, or is sufficient. Otherwise, go to step 2, increasing .

Where is called the acceptance threshold and expresses the minimal improvement of the gain function that the new solution has to provide in order to be chosen. It may be useful to relate the to the estimate of the standard deviation () of the gain function evaluations. If the new solution’s improvement is, for instance, twice as great as , and the noise is at least approximately normally distributed, then it is rather unlikely that the difference in the value of the gain function is just the consequence of the noise. It is far more likely that the new solution is indeed better than the current one.

Unfortunately, none of the above approaches is perfect. The first one explicitly requires a significant increase of the number of function evaluations, whilst the second one will sometimes discard a better solution that because of the noise will have too low gain function value to get over the threshold . As a result, the second approach will also require more function evaluations. In practice, the noise sometimes depends on the solution that is evaluated or has a different distribution, which makes each of the above methods’ performance strongly problem-dependent.

### Hill-climbing

The method of hill-climbing is one of the oldest optimization techniques. Nevertheless it is also one of the most popular deterministic approaches to optimization, being also a good starting point for the further improvements [25].

Hill-climbing, similarly to a basic random search method, is a greedy search algorithm. The greedy algorithm is one that always chooses the best solution in the current situation, not caring about the consequences in the long run. In case of the hill-climbing algorithm, its greedy approach in consequence may cause it to finish the search at local maxima, therefore it is a local optimization algorithm, as explained in Subsection 2.2.1. However, in many cases hill-climbing can provide a good solution in shorter time than other global search algorithms [4].

When hill-climbing chooses the next solution , it analyses all the vicinity of the currently best known solution , defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a dimensionality of the vector and is a vector of elements with at the position and zeros elsewhere. Therefore, is a set of all solutions that differ from the current one in exactly one vector element, by either or , where is called the step size.

The actual hill-climbing algorithm can be defined as follows:

Algorithm : Hill-climbing

1. Randomly choose an initial solution . Calculate .
2. Choose
3. If then let , otherwise stop and return .
4. Go to step 2, increasing .

We can notice that the algorithm performs an optimization process within itself, searching for the best . This is basically performed by evaluating every vector in separately, but, depending on the problem, it is often possible to improve performance by modifying this step.

When the hill-climbing algorithm finds no improvement in the , it terminates. Another possible option would be to decrease the step size . This would cause the algorithm to search more carefully close to the location of the currently best known solution.

Noisy gain functions

When it comes to the hill-climbing algorithm and the noisy gain function information, methods that can be applied are analogical to those described in Subsection 2.2.3 for the case of a localized random search: gain function evaluations averaging and acceptance threshold.

Unlike localized random search, hill-climbing is a local search algorithm trying rather to find the closest local maximum, which may allow it to find a good solution in relatively shorter time. Therefore, if the number of evaluations is very high, as in the case of optimization with a noisy gain function, the hill-climbing algorithm may prove to be quite effective.

### Finite-difference stochastic approximation

A great variety of deterministic optimization methods exist for problems with differentiable gain functions. Many of them are based on the gradient of the gain function:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

One of such methods is called gradient ascending. Knowing the gradient of the noise-free gain function, we can find the local maximum by following the algorithm below:

Algorithm : Gradient ascending

1. Randomly choose an initial solution . Let .
2. Let . Let .
3. If is sufficient or then stop and return . Otherwise go to step 2.

The *step size* is can be a constant or an element of a predetermined sequence, or the solution to the secondary optimization problem: , called the *line search* [25].

Although the above algorithm is deterministic, many methods of stochastic optimization are inspired with it. The idea of the algorithm is simple: at each step it modifies each element of the vector in a way that moves the element towards the increasing values of the gain function by the step , basing on the gradient . For instance, if the gradient is positive for a given element, it means that the gain function grows as the value of the given element increases. Therefore the element’s value will be increased.

Noisy gain functions

A stochastic version of Algorithm 4 exists that can be used in the case of the noisy gradient functions. Many times, however, we do not even have a noisy gradient function – we may only have a noisy gain function. In those cases, we can use a gradient estimation basing on the noisy gain function instead. In order to do so, we need to use one of gradient estimation methods.

A simple but effective method for gradient estimation is called a *finite-difference* method. Let’s denote the gradient estimation for step and the solution with . The finite difference gradient function estimation is defined as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a noisy gain function, is a vector of elements with at the position and zeros elsewhere and is called a difference magnitude.

The parameter determines the sampling of when estimating its gradient. The finite-difference gradient estimation method can also be used in its one-sided form, when we sample just on one side of . In the one-sided form, the -th row of the gradient estimation would be given by .

The following algorithm, called a finite-difference stochastic approximation, differs from Algorithm 4 only in the fact that it uses the gradient estimate instead of the gradient function. However, it allows us to apply the algorithm for a wider range of problems [25].

Algorithm : Finite-difference stochastic approximation

1. Choose randomly an initial solution . Let .
2. Let . Let .
3. If is sufficient or then stop and return . Otherwise go to step 2.

The parameters and from (2.15) are called gains of the algorithm. The convergence theory of the finite-difference stochastic approximation algorithm to a local maximum expresses these parameters as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where , , and are strictly positive constant parameters, is called a stability parameter and is a number of algorithm iteration.

Under certain conditions and an appropriate choice of the above parameters, the algorithm will converge to local maximum. However, in practice we can often use less rigorous gain setting than the one required by the convergence theory, which may prove to be more effective. In [25] the following approach is proposed to choose the values of the gain parameters:

Algorithm : Semiautomatic gain parameters setting

1. Initially set and
2. Estimate the standard deviation of the noisy gain function. Set .
3. Choose to be significantly lower than the number of iterations that are planned to run, for instance .
4. For each choose such that is approximately equal to the desired size of the step for element of the vector . Choose .

In step 2, the standard deviation can be estimated on a possibly large number of evaluations of a constant solution . The setting of to be significantly smaller than in step 3 is justified by the fact, that when reaches its higher values it will have a greater influence on the value of then , making sure that the step size is higher at the beginning of the optimization and lower when the optimization finishes. Finally, in order to measure the gradient estimation reliably using noisy gain function in step 4, we need to perform more measurements and use the average result instead.

The finite-difference stochastic approximation is a local optimization algorithm, which at each iteration tries to improve the current solution by adjusting every element of the vector with the use of the gradient estimation. On the other hand, the hill-climbing algorithm (Section 2.2.4), adjusts just one element of at each iteration step. The finite-difference stochastic approximation method also reduces the step size at each iteration, allowing the search to focus on a smaller area close to best known current solution .

### Simulated annealing

In this subsection, a global optimization method basing on the annealing principle is outlined. Essentially, annealing algorithms are reducing the magnitude of random perturbations used in the search process in a controlled manner. This helps them to avoid converging to a local maximum in the early stages of the search process.

The annealing principle and its name derive from the analogy to the process of controlled cooling of substances. At high temperatures, the molecules of the substance move chaotically, having a high energy level. As the temperature decreases, the molecules slow down and at one point they may start to align in a crystalline structure, which is their minimal energy state. However, if the rate at which the temperature decreases is too high, an amorphous state may be reached which is not the desired minimal energy state.

In analogy to optimization, the lowest energy state is the global optimum we are searching for, and the temperature is the factor that specifies the magnitude of perturbations that appear while searching for the optimum. In statistical mechanics, when the substance cools down, it is possible for it to sometimes increase its energy level, as an effect of a random perturbation. Similarly, in the annealing principle-based optimization algorithm it is possible that sometimes the worse solution is chosen instead of the better one, especially at the beginning of the process, when the temperature is high. This increases the search space exploration of the algorithm, which may result in finding the global optimum [26].

Metropolis criterion

Basing on the annealing principle and the Boltzmann-Gibbs probability distribution, the Metropolis criterion was proposed to use in numerical analysis. The criterion states that if the system is at energy state , and in result of a random perturbation it is possible for system to move to another energy state , it will move to always if . Otherwise, if , the probability of the system moving to the is given as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is Boltzmann’s constant and is the system’s current temperature.

Therefore, in the Metropolis criterion, if the new solution has a higher energy state than the current energy state, there exists a non-zero probability that the system will move to the higher energy state. Using the Metropolis criterion modified for the maximization rather than minimization problem (to keep consistency with the other algorithms described in this chapter), we can outline the algorithm called simulated annealing*[[3]](#footnote-4)*:

Algorithm : Simulated annealing

1. Choose randomly an initial solution .

Set an initial temperature . Calculate .

1. Choose randomly an independent vector , where is the dimensionality of . Let , make sure that .
2. Calculate . Let .
3. If , let . Else if and random variable uniform on satisfies , let . Otherwise, let .
4. Repeat steps 2, 3 and 4 until the budget of gain function evaluations allocated for the given temperature has been used.
5. Stop if the maximum number of evaluations has been reached, or is sufficient. Otherwise, go to step 2, increasing and decreasing the temperature according to the annealing schedule.

We can notice that at step 4, the Boltzmann constant was skipped, as we assume that we have a full control of the temperature . Also, in step 5 the ‘budget’ allocated for each temperature value is introduced. This allows us to choose how often the temperature should change.

When using the simulated annealing algorithm it is important to choose the appropriate cooling schedule, i.e. the sequence of the temperature values. There are many possibilities, like decreasing the temperature geometrically , where , or on per-iteration basis, for instance: . Typically, we need to analyze how the system behaves during a small number of evaluations and try to set these parameters appropriately, using a trial and error method.

Another important aspect that strongly influences the effectiveness of the algorithm is the distribution using which is chosen. It should meet the same requirements as in case of localized random search (Algorithm 1) – the mean should be zero and the magnitude of the perturbation of each element of should be consistent with the range that a corresponding element of stays within.

Noisy gain functions

When the gain function contains random noise, the methods that can be applied to minimize the impact of the noise are similar to those described in Subsection 2.2.3 for the case of localized random search: gain function evaluations averaging and acceptance threshold.

Whilst the gain function evaluations averaging can be introduced without any problems, the acceptance threshold requires us to change Algorithm 7 a little more than just replacing the with - we need to modify the way that is calculated in step 3:

Algorithm : Simulated annealing for a noisy gain function

1. Choose randomly an initial solution .

Set an initial temperature . Calculate .

1. Choose randomly an independent vector , where is the dimensionality of . Let , make sure that .
2. Calculate . Let .
3. If , let . Else if and random variable uniform on satisfies , let . Otherwise, let .
4. Repeat steps 2, 3 and 4 until the budget of gain function evaluations allocated for given temperature has been used.
5. Stop if the maximum number of evaluations has been reached, or is sufficient. Otherwise, go to step 2, increasing and decreasing the temperature according to the annealing schedule.

The value in this case can be either positive or negative. When it is negative, only a new solution that has a significantly higher noisy gain function value will satisfy the condition and will be chosen as the new best known solution. Otherwise, if , the worse solution will be chosen with less probability. Negative values may be useful in the case of the gain functions in which we suspect the number of local maximums is low – we are less willing to give up our best known current solution, unless the gain is big enough.

On the other hand, a positive value of can be used when we suspect the gain function to have many local maximums. In that case, the new solution will be chosen if its noisy gain function value is even a little lower than the current solutions’ gain function value. Otherwise, the new solution may still be chosen with a little higher probability than in case of .

In both cases it is useful to set , where is a standard deviation of the noisy gain function.

## Quake II

Quake II is an FPS game released in December 1997 by ID Software. It has been extremely well-received, selling over one million copies and becoming the most popular online game for all of 1998 [27]. Four years after the first game version was published, the complete source code has been released under the terms of GNU General Public License, which has created an opportunity to use the game, for instance, in research.

Since then, the game has been used in many different experiments. One of them was even involving the use of a modified version of Quake II as a virtual reality environment for a laboratory mouse during the study of how the brain's hippocampus creates maps of its surroundings [28].

The game itself has been designed to be a challenging entertainment for humans; therefore it requires a full spectrum of reactive, tactical, and strategic behaviours from a player, which is one of the factors that make Quake II attractive for AI researchers [12]. Furthermore, we can directly observe the interaction of our agent with human players during a game-play, which gives us opportunity to conduct experiments like BotPrize mentioned in Section 1.5.

### Quake II deathmatch

The basic game mode in Quake II is so-called deathmatch. It is a game performed in a particular three-dimensional environment called a map, in which the goal of each player is to eliminate as many opponents as possible, while staying alive for as long as possible. After a player is defeated, he or she respawns back into the world, simply appearing alive again at one of spawn points on the map. Each time a player defeats an enemy, he or she receives a point, called a frag. At the end of the game, the player who has scored the most wins. To defeat the enemy means to inflict such damage that in consequence, the enemy *dies* (and then immediately respawns back to game). The player dies when his or her health level drops to 0 or below (initially, after respawn being 100). During the game players can pick up items which are distributed on the map. There are four main categories of items that a player can pick up and use (Figure 7):

* *Weapons* – used to fire at enemies. There are 11 types weapon that differ with a damage they inflict, the ammunition they require, the dispersion of projectiles, and gun reloading time.
* *Health* – used to recover from wounds by increasing the player’s health level. There are different types of health items that provide different health benefit.
* *Armour* – reduces the damage the player receives from enemies. Generally, armour behaves similarly to additional health. When the player receives damage from the enemy gunfire, first his or her armour level is decreasing. After it reaches 0, the health level starts to decrease.
* *Ammo* – the ammunition that is required to use weapons. There are 6 types of ammunition in Quake II.

Items also respawn after they have been picked up by a player, but not immediately – each item reappears after some specific threshold of time, different for each item type.

|  |  |
| --- | --- |
| quake002.png quake003.png | quake02.png |

Figure 7: An example of Quake II game items and opponent.

### Quake II client-server communication

The game uses a client-server model for communication. One of the game participants starts a server to which others connect. After that, the server enters the server loop, during which it receives messages with actions from all game participants, executes them in its internal game environment representation and sends a world state update to each client. Each client updates its current game state, displays it to the player, and reads a player input from keyboard and mouse, which is interpreted and forwarded to the game server as player’s action.

The server sends updates of the world state to all its clients as often as every 100 milliseconds, using its own application layer protocol based on the UDP network protocol. Due to efficient client server communication, the players can play Quake II in a real-time manner using relatively slow network connection.

## QASE API

Quake II Agent Simulation Environment – QASE is a comprehensive, feature-rich Java API that provides functionality that facilitates high-end AI-related research as well as its use for educational purposes. The authors of QASE also intended to foster further interest in the adoption of commercial computer games in academic AI community [12].

Among many features of QASE API, the most important are:

* Full access to current game state information (perception).
* Easy to use interface for controlling bot’s behaviour (actions).
* A set of tools for handling bot’s waypoint-based map representation (as described in Subsection 2.1.2) – creating a waypoint map basing on pre-recorded data of the human player movement, accessing it, finding shortest paths and modifying it.
* A Binary Search Partition Tree parser, used to access environment’s spatial representation from Quake II’s resource files. This is mostly used for collision detection, which is not possible to perform using a waypoint map only.
* Integration with MatLab® environment.

Using QASE API allows us to focus on our research task instead of the development of simulation environment or the integration issues.

Bots developed using QASE API are client-side bots. As mentioned in Subsection 2.3.2, Quake II server sends new messages to each client every 100 milliseconds in order to keep all the clients up to date with the world information. As the server informs clients only about the world in their immediate surroundings, client-side bots have to deal with limited knowledge. On the other hand, the server-side bot can have an access to the full knowledge about the current game state. This includes positions of all the enemies and availability of the items even in the most remote parts of the map, which gives server-side agents an important advantage over client-side bots.

QASE API is technically incompatible with some of existing, third-party Quake II bots that could be used for the comparison with developed solutions. To be precise, some of the bots implemented on the server-side are not correctly recognized by QASE API, remaining invisible for every bot that is implemented using it. Fortunately, one of the most popular bots – EraserBot (described in the following section) works without any known issues with bots developed using QASE API, therefore it will be used in this thesis.

## EraserBot

Since the Quake II game was first released in 1997, the number of the Internet users has grown over 28 times [29]. It is not surprising that many players who did not have constant access to the Internet in that time were using artificial players that could be played against without the network. Bots were quite popular way of exercising and playing without other human opponents. There are over 14 popular Quake II bots. EraserBot by Impact Development Team is one of them, by many considered to be the best available Quake II agent [30]: “Of course nothing can ever replace playing those real, human opponents; but the Eraser bot comes as close to simulating that as anything can.”.

Among many other features, EraserBot is capable of learning the navigation on new maps from human players’ example, it has configurable skill levels allowing users to adjust the “character” of the artificial player, like aggressiveness and accuracy and, when playing in teams, it can receive simple commands from other team mates, like grouping in order to attack the enemy together.

EraserBot will be used in this thesis in order to compare it with the developed solution.

# Developed solutions

This chapter demonstrates all crucial decision making algorithms that have been used by the developed agent. After a short explanation of how the agents are created with the use of QASE API, the description of the knowledge representation used by the agent is given. Next, the main algorithm concept is sketched, following with a detailed explanation of the most important steps of it, concerning bot’s navigation and firing.

## Creating a bot using QASE API

Using the features mentioned in Section 2.4, the implementation of the most basic agent is quite straightforward. The QASE API calls the runAI method of our bot implementation, assuring before that the new game state is available. In the method itself, we can conduct necessary computation and set bot movement, firing or any other available action that will be performed in the next step. We need to perform all our computation within a limited time called a *frame of execution*, i.e. the interval time between game state updates received from the server. If our method does not finish on time, the QASE will not send the new message to the server and the server will assume that we repeat the actions that we have performed in the previous frame.

The content of the runAI method will determine the entire behaviour of our agent, so all we need to do now is to implement the algorithm that will keep our agent alive for as long as possible, while eliminating as many opponents as it can. Nevertheless, as QASE API authors state, implementing such an algorithm is “quite a challenge both from the perspective of autonomous agents and the perspective of artificial intelligence” [12].

## Knowledge representation

In order to store and access the knowledge about agent’s environment efficiently, we need to use an appropriate world knowledge representation.

Waypoint map

One of very useful features of QASE API is a tool called waypoint map generator that allows us to create a waypoint map from pre-recorded demo files. Demo file is a file that can be created during any game of Quake II, containing the full record of player’s activity during the session. Essentially it is a copy of the network stream received during a game. It is often used to demonstrate player’s skills or as a proof of the result of a particular match or a tournament. However, we can also use a demo file to generate a waypoint map for our Quake II agent.

In order to do so, the waypoint map generator gathers all the positions in the world that have been occupied by the player recorded on the given demo. Next, these positions are clustered using k-means algorithm to produce a smaller number of waypoints. Finally the connections between these waypoints are added basing on movement of the player recorded on the demo, creating a full waypoint map that can be used by our agent. Due to the factor that the geometry of the environment in Quake II does not change, the waypoint map is rather static. Therefore, the Floyd’s algorithm is used to pre-compute the shortest paths between each pair of waypoints [12]. In addition, each waypoint can store information about the item that can be found next to it.

Enemy information

Further important information that the bot needs to store is the information about the opponents. Many times during the game, the enemy disappears from our field of view. In this case it is more likely to find the enemy going towards the point where it was seen last time. This is used, for instance, to avoid the enemies while planning the path to the item the agent needs to pickup. The agent updates available enemy information every execution frame. If the information is older than the specified threshold time, it is discarded.

On the other hand, when the enemy is visible, we want to know not only its current position, but also the position that it has occupied in the previous frame. This can be very useful when trying to predict opponent’s future position, as described in Section 3.5.

Items

Knowing that items, after being picked up, reappear or respawn at the same place on the map after some known time, it is useful to store information about the last time when the given item has been seen at its spawn position.

If the agent doesn’t see the item at the expected position, it can estimate when, in the worst case, the item should reappear there. This is very useful when choosing which remote item the agent should go to in order to pick it up.

Further information that should be stored is whether the given item is reachable for the agent or not. Not all the items that the bot can see can be picked up by it, as some of them may be placed, for instance, behind a wide gap that cannot be jumped over by the agent. The waypoint map itself does not allow us to check if the given item is reachable. To do so, we need to access the world geometry information stored in a Binary Space Partition Tree[[4]](#footnote-5) provided by Quake II for each map and check if there is actually a walk-able surface between our current position and the item we want to pick up. Although the Binary Space Partition Tree is known for its good performance, it is computationally expensive to calculate whether an item is reachable or not, every time we see it. Therefore, this information is kept in agent’s knowledge base.

## The main concept

Basing on observations of human players, the author concluded that most of successful players, apart from having high firing accuracy during combat, are constantly picking up items that give them advantage over their opponents. This observation has been used as an initial concept to develop the agent here. At every moment of the game, the bot is either on its way to pick up some item or it chases the enemy. This is done by establishing so-called navigation plan by choosing the item the agent wants to pick up, obtaining a path using the waypoint map and starting to follow it. In the meantime, the bot’s combat module establishes firing decisions and shoots at visible enemies.

In this form, the algorithm constantly tries to improve agent’s inventory, the health and armour state, at the same time taking every possible chance to inflict damage on the enemy. The following part involves the basic explanation of the six most important steps of the developed bot’s algorithm taken at each execution frame:

Algorithm : Agent’s main loop

1. Update the knowledge base – this step updates the information described in Section . with new information about the world given to the bot at the current frame.
2. Establish navigation plan – in this step either the old navigation plan is continued or the new plan is established. If the new plan needs to be established, the destination item is chosen basing on bot’s current state and the situation in the environment. Then the path from the current bot’s position to the chosen item is found using the waypoint map.
3. Get navigation instructions – basing on a currently executed navigation plan, the path that it provides and the bot’s current position, the direction of the bot movement for the next execution frame is computed.
4. Establish firing decision – in this step, the bot decides using which weapon and at which visible enemy it should fire. If there are no visible enemies, the firing decision will be empty.
5. Get firing instructions – basing on its own and the enemies current positions, the weapon the bot decides to use and its characteristics, the bot calculates the angles at which it should fire the gun in order to hit the opponent.
6. Execute instructions – the movement and firing instructions are passed to QASE API which sends them to the game server.

In the following sections, the most important steps – 2, 4 and 5 are described in a more detailed manner.

## Navigation plan

The navigation plan essentially consists of a destination item and the path that the bot needs to follow to reach it. When the agent establishes the navigation plan for the currently processed execution frame, it has to decide first whether to find a new plan or continue with a current plan. The bot decides to change its plan when:

* There is no current plan
* The old plan is accomplished
* The agent is stuck for some reason (not moving for some period of time)
* The execution of the current plan has reached its time limit

If one of above conditions is true, the new navigation plan will be created.

The kind of a plan described above is called a regular plan. Due to a dynamic nature of the game, it is important to reconsider the plan often enough, to make sure that our agent is responsive to changes of its state and environment. At the same time establishing a new plan at every execution frame could lead to indecisive behaviour, where the agent changes the decision too often. To address that, each plan has a time limit that makes sure that it will be reconsidered after that particular time passes. After the time limit has passed, a new plan is being created, but in most of the cases, the new plan will have the same destination item as the previous one. This is because the algorithm tries to choose the best available destination item, and unless something important has changed in bot’s or environment’s state, the same destination item should be chosen.

There can also appear so-called spontaneous plan. These plans are created when the agent is in the middle of the execution of a regular plan, and there is a good opportunity to pick up some item – it is reachable and close to the agent’s current position. In that case, the spontaneous plan is created and the agent executes it for a short period of time, picks up the desired item and then continues with the previous regular plan. This allows our agent to use the opportunities when, for instance, the opponent drops a weapon somewhere close to the path that our agent follows.

Another kind of a plan is called the enemy engaging plan. This kind of plan is established instead of a regular plan when our agent’s state is considered to be good enough to attack the enemy. If there are no enemies to attack, the regular plan is created. How one shall understand that the bot’s state is good enough to attack the opponent? This question will be answered in Subsection 3.4.3, after explaining how the agent chooses the destination item for a regular navigation plan.

### Criteria for choosing destination *item*

When the agent decides to create a new regular plan, it creates a ranking of all known items that can be picked up at the moment, where each item has assigned a fuzzy membership value that expresses the degree at which the bot wants to pick up given item. The item with the highest value is chosen. All the factors that are used in the formula for calculating each item’s fuzzy value are explained below:

Bot’s health deficiency

Bot’s health level value can range from 0 to 100. The health level deficiency is calculated using the following formula:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is bot’s health level in state . The health deficiency is high when the agent needs health and low, when the agent has enough health.

Bot’s armour deficiency

Armour level value can range from 0 to 100. The armour deficiency is calculated similarly to the health deficiency:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is the agent’s armour level in state .

Bot’s weapon deficiency

Each weapon has assigned a certain weight. The more effective the weapon is, the greater its weight should be. There are eleven different weapons. Bot’s weapon deficiency measure should be close to 0 when the bot owns many good weapons and close to 1 when the bot owns just a few, not very effective weapons. The bot’s weapon deficiency is calculated as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a set all weapons owned by the bot in state , is a set of all the weapons available in the game, and is a weight of a weapon .

Bot’s ammo deficiency

Each weapon uses one of 6 types of ammunition available in the game. For each of those types there is the maximum amount of ammunition a bot can carry. Analogically to other deficiencies, we want bot’s ammo deficiency to be low when the agent owns a lot of ammo. To calculate ammo deficiency the agent uses the following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a set of all weapons owned by a bot in state , is a set all the weapons available in the game, is a weight of a weapon , is bot’s level of ammo for an ammunition that is used by a weapon in bot’s state and is a maximal possible level of ammo for an ammunition that is used by a weapon .

Item’s pickup benefit

Each item influences one of the 4 characteristics of a bot state that were described above: health, armour, weapons or ammo. The pickup benefit of an item of category is a difference between current bot’s deficiency in category and the deficiency the bot would have after picking up that item. This value is in addition standardized by dividing it by the highest such value in the category , which assures that all the values will be in the range of :

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is an item that we calculate pickup benefit for, is an item category to which belongs to (one of described earlier: health , armor , weapon and ammo ), is current bot’s state and denotes bot’s projected state after picking up the item . Where is the set of all available items of category that the bot considers in state .

Distance factor

The distance to the item is measured following the shortest path from bot’s current position to each item that we consider. Next, the longest distance is chosen. The distance factor for a given item is a distance following the map to the item divided by the distance to the furthest item. This standardizes the distance factor, like all other measures in a range . The final distance factor for a given item can be expressed with a following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is bot’s state, is an item we calculate the distance factor for, is a set of all the available items, and is a distance following the shortest path on the map from bot’s position at the state to the item .

Enemy cost

This measure expresses the possibility of encountering the enemy on the path towards the chosen item. It is estimated by calculating the sum of waypoint risk measures at each waypoint that is a part of the path from bot’s current position to the item. The waypoint risk measure can be expressed as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |
|  |  | (.) |

where is current bot’s state, is the set of all the enemies that the bot knows about at the state , is an Euclidean distance between positions of the waypoint and the enemy and is some constant, used as a threshold.

The enemy cost metric for an item is a sum of all waypoint risk measures for all the waypoints that are a part of the shortest path from current bot’s position to the item, divided by the maximum enemy cost metric for considered set of items, in order to standardize the result. This can be described with a following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is the item, is a shortest path from current bot’s position to item and is a waypoint on the path.

### Fuzzy logic application

The measures introduced above can be perceived as values of membership functions of fuzzy relations, as all of them are within the range and they all express some logical relation, as listed in . Calculating these values, the agent performs a fuzzification – mapping of the numerical input data to fuzzy degrees of membership of each relation.

|  |  |  |
| --- | --- | --- |
| Fuzzy value | Symbol | Represented relation |
| Health / armour / weapons / ammunition deficiency |  | Bot needs health / armour / weapons / ammunition |
| Item’s pickup benefit |  | Picking up given item is beneficial for a bot |
| Distance factor |  | Item is far from bot’s current position. |
| Enemy cost |  | The path to the item is dangerous for a bot. |

Table 1: Logical statements that are expressed by each of introduced fuzzy variables

Now, the agent can apply fuzzy rules and then establish bot’s final navigation decision. Looking at available logical statements and their fuzzy representations in Table 1, we can come about with a following rule for choosing best item. Given bot state and item belonging to category :

IF bot **needs** items belonging to category   
AND picking up item is **beneficial**   
AND path to item is NOT **dangerous**   
AND item is NOT **far**  
THEN bot should pick up item .

Let’s denote the conclusion of the rule above, the fuzzy relation expressing logical statement “bot at state should pick up item ” with . The fuzzy value of the is a conjunction of four other statements. Applying Yager’s intersection operator from Definition 4, we can express the above rule with a following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is bot’s state, is an item bot wants to pick up and is item’s category. The complement operator is defined as in Definition 2, and the Yager’s intersection operator is defined as in Definition 4, with parameter .

Using fuzzy relation a bot can choose the item that has the highest fuzzy membership value at each bot’s state , when agent needs to create a new regular navigation plan. However, there is a problem with a defined as in (3.10): the importance of each relation in the intersection is the same. To avoid it, the weights for each relation have been introduced in order to allow us to tune the relation and decide, for instance, whether the relation “bot needs health” is more important than “item is far from bot”. The final form of follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a weight for bot deficiency in category , is weight for benefit of picking up items in category , is weight of distance factor (i.e. complement of relation) and is a weight for the safety factor (i.e. a complement of ).

Therefore, there are 10 different weights that need to be adjusted appropriately in order to get the best result using the relation: deficiency weights for each category: , , and , benefit weights for each category: , , and , distance factor weight and safety factor weight . Each weight needs to belong to range in order to be used with fuzzy membership functions and operators.

### Enemy engaging plan

As mentioned earlier, when agent’s state is considered to be good enough, the enemy engaging plan is created that will lead the bot to a random position close to the enemy in order to attack it. The quality of current bot’s state can be expressed averaging bot’s weighted deficiencies for each item category:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

When bot’s state is considered to be good – the bot’s deficiencies are low, the is close to , whilst if bot’s deficiencies are high, the is close to 1.

Now, if at current bot’s state there are known positions of the enemies, and the value of is lower than a parameter called aggressiveness (denoted ), the bot will create a new enemy engaging plan leading to the surroundings of the closest enemy. If the weights are adjusted properly, the agent should search for the enemies only when it’s state is good enough to enter into the fight.

Having chosen the navigation plan, the agent can now follow the established path by turning itself towards the next waypoint on the path and moving forward. Once the waypoint is reached, the bot moves towards the next waypoint from the path.

## Firing decision and instructions

In this section, the steps 4 and 5 of Algorithm 9 are described – the establishing of the firing decision and computing the firing instructions that will be passed to QASE API.

### Weapon and enemy choice

In the early stages of work on this thesis, the task of appropriate weapon choice according to the distance to the enemy was considered.

In Quake II weapons differ with the dispersion of their projectiles which gives an impression that the weapon with high dispersion will be effective only in close distance. On the other hand, another weapons’ projectiles move relatively slower than others’, which may cause poor effectiveness when firing at further, moving target. Surprisingly, the conducted experiments have proven that the simple strategy of always choosing the best weapon the bot possesses gives best results. One of the reasons for that may be a good aiming skill the bot has or existence of weapons that have both - low projectile dispersion and high projectile speed. Therefore, the weapon choice algorithm bases on the weights assigned to each weapon – , where is a weapon item. These same weights where used in equations (3.3) and (3.4). The bot always chooses the weapon with the highest weight it owns.

The enemy choice has also been proven not to be a very complex task. Out of visible enemies in most of cases it is best to choose the enemy that is the closest, as it is easier to aim at and hit and at the same time he represents a higher threat, as he can damage us more than the enemy that is further.

### Aiming algorithm

When the weapon and the enemy to shoot at are chosen, the agent needs to establish where exactly to shoot, in order to hit the running opponent, often using a weapon which fires projectiles that do not move fast enough, to hit the enemy immediately. Actually, out of 11 weapons available in the game, just 5 hit the target immediately. The rest fire a projectile that moves with a certain speed different for each weapon. Using such weapons requires so called leading – aiming not directly at the enemy’s current position, but at the position we predict it to be at when the projectile will reach it. This task requires a lot of skill from human players. An expert human player can be incredibly accurate at hitting the enemy using a leading technique, aiming and firing accurately in a fraction of a second.

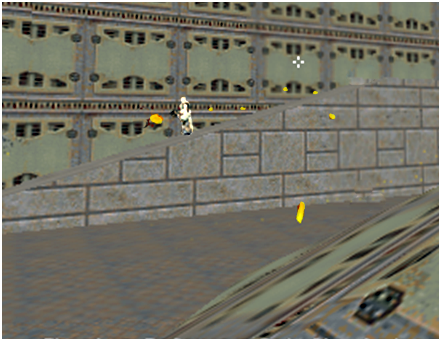


Figure 8: An example of *leading* technique. The agent aims at the point marked with a white crosshair. The yellow projectiles reach the enemy with a delay, when he will already change his position. Therefore, the agent needs to aim at the position, where it expects the opponent to be when the projectiles will reach it.

In order to teach our agent to aim using a weapons firing slower projectiles, we need first to predict the position of the enemy. One simple and quite effective way to predict enemy position is to observe its current velocity vector , by comparing its current position with a position one execution frame before. As experiments show, quite often the enemy continues to move with the same velocity, which allows us to predict its future position, using linear regression-like approach, taking into account just one position back. Using more than one position back resulted to be less effective. Knowing the enemy velocity and the speed of the weapon projectile , we search for a vector - the direction of our shooting, as shown in Figure 9.

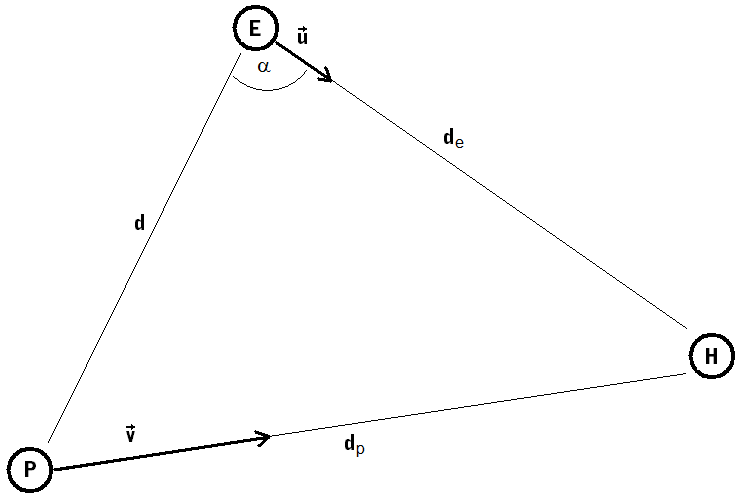


Figure 9: The illustration of the aiming task on a plane. Where is a position of the shooting agent, is a position of the enemy, is the enemies’ velocity and is velocity of a fired projectile that we are seeking. If the enemy will continue to move with the velocity , the projectile will hit it at the point .

Assuming that the projectile will hit the target after time , the distance travelled by the enemy will be , and by the projectile . We also know the angle marked in Figure 9: . Using this information and applying the law of cosines to the triangle from Figure 9, we can formulate the following equation:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Using which we can calculate the time , which in turn allows us to find the exact position and calculate the vector - the direction at which the agent will fire.

## ReferenceBot and LearnBot

The 11 weights described in Section 3.4 will be used in the following part of the thesis as variables that will be adjusted in order to optimize bot’s performance, as a vector from Definition 5.

The bot that will be having its weights adjusted will be called LearnBot, whilst the agent whose weights were tuned manually by trial-and-error method and are constant throughout whole the thesis will be called ReferenceBot.

# Experiments

In this chapter the experiments that were performed are described and their results are presented. There are three basic parts of the experimental procedure: the initial experiment, the optimization of bot’s navigation module parameters and the evaluation of found solutions. In the following sections each of these parts is described, the parameters required for each experiment are chosen and the results are presented and compared.

## Initial experiment

The aim of the initial experiment is to obtain data that will allow us to compare the results before and after the optimization of bot’s navigation module parameters in order to observe the potential improvement.

### Experiment description

The bot’s navigation module weights setting that will be evaluated during the initial experiment has been manually adjusted during the development process of the program. These setting bases on author’s intuition and has been evaluated in the game mainly by the author himself. It will be used as a setting for ReferenceBot.

In the initial experiment, ReferenceBot will be compared with EraserBot. There will be 5 separate deathmatch games played, each of 30 minutes on the same map. One ReferenceBot will play against one EraserBot. The skill level of EraserBot will be set to the highest possible. The same thing will be done with the characteristics of EraserBot, such as firing accuracy and combat skills. Each game’s result will be calculated as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is a number of times when the given bot has eliminated the enemy, and are ReferenceBot and EraserBot, respectively.

The map that will be used to play the game on is probably the most popular Quake II map, called q2dm1, illustrated in Figure 10.

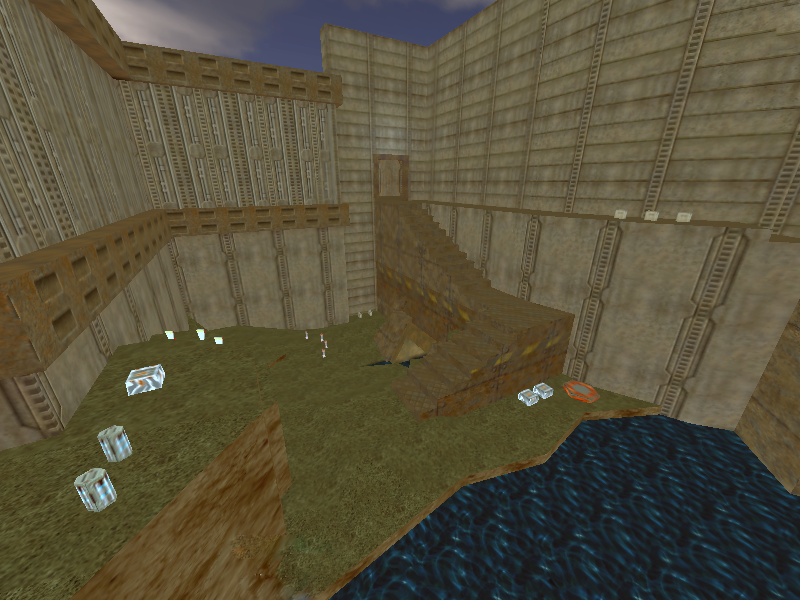


Figure 10: An overview of the part of *q2dm1* map that will be used in initial experiments.

### Results

In the initial experiment, EraserBot proves to be more effective – the average result is . The scores of each bot are presented in Figure 11.

init2.emf

Figure 11: The average score of each bot in the initial experiment. ReferenceBot uses   
the manually adjusted parameters.

During the game it has been observed that EraserBot would often win using one of the weapons that was very rarely used and picked up by ReferenceBot. EraserBot also explored the map more, getting to more remote and useful items. This may be due to better world map representation or ReferenceBot’s navigation weights not adjusted properly.

## Optimization

This section describes the bot’s optimization procedure that has been applied. During the optimization, LearnBot’s weight parameters that are used when establishing the navigation plan, as described in Section 3.4, will be changed to possibly maximize bot’s result in the game against ReferenceBot, which will use a constant, manually adjusted configuration. Our gain function that we are maximizing can be described as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

where is the configuration we are evaluating and is LearnBot using the configuration .

Each Quake II game that will be run during the optimization process will be a duel of LearnBot against ReferenceBot, i.e. one LearnBot will play against one ReferenceBot. LearnBot’s configuration will be adjusted according to given optimization algorithm.

In order to make optimization more efficient, a map designed on purpose will be used. It is smaller than standard Quake II maps, which will allow agents to play more dynamically[[5]](#footnote-6). It was designed to provide equal chances for each player, regardless where he or she will appear after being defeated by the opponent. The relatively simple geometry of the map allows us to run the game at the maximal speed – in case of complex standard maps that are provided together with the game, the optimization process would have to be at least 2 times slower. Items on the custom map are distributed in a symmetric manner. Weapons, ammunition, health and armour are separated, so the agent needs to make a good decision about what is more important for it when deciding where to move. More powerful weapons and useful items are placed in more exposed locations, so it may be more dangerous to try to pick them up. The simplified plan of the map is illustrated in Figure 12.

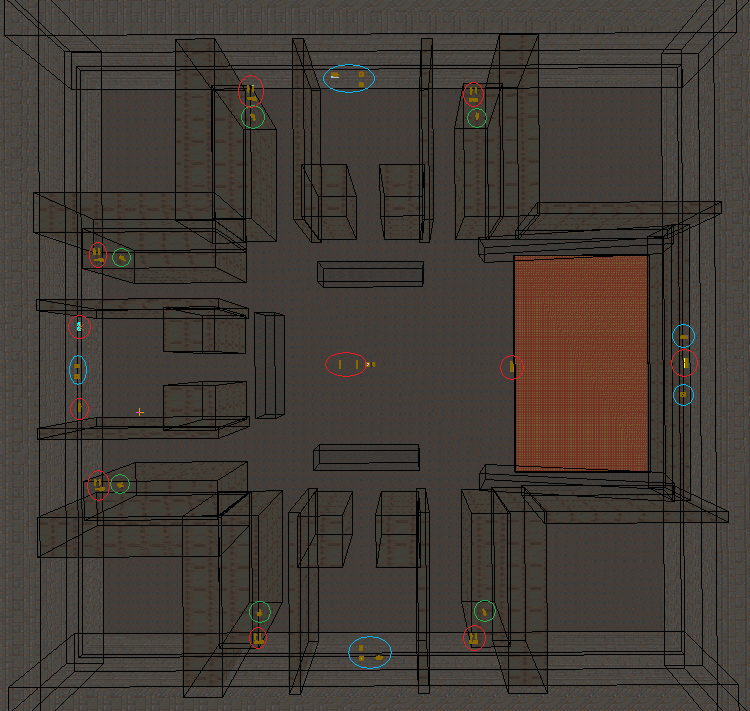


Figure 12: The overview of the map that will be used during the optimization process. The locations marked with green are player’s *spawn* or “reappearance” locations. The blue locations are those with health and armour items and the red ones are with weapons and ammunition.

### Results measurement

The Quake II game provides a complex, real time, three-dimensional environment. In game aspects such as weapon fire inaccuracy or the exact position of player respawn, the game introduces randomness. What is more, each agent has information only about the state of its immediate surroundings, which is not sufficient to determine the behaviour of the enemy even if we knew the exact, deterministic algorithm it uses. For instance, we do not have information about the level of health of the opponent we see. We don’t know if it is high, or low because of the fights with other opponents it took part in.

Because of the game complexity, each game’s result may be different although exactly the same agents were playing. This can be interpreted as noise that is added to our gain function as defined in (4.2). Therefore, we only have an access to the noisy measurement of the function:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Each measurement of the function is equivalent to one Quake II game.

To account for the noise, we will have to use stochastic optimization methods and, in a result, drastically increase the number of games we need to run in order to improve agent’s performance.

First, we will apply the method of averaging of the measurements – we introduce the function that will average on measurements of . We can denote it as follows:

|  |  |  |
| --- | --- | --- |
|  |  | (.) |

Although this decreases the influence of the noise, it is not our goal here to remove most of it using just this method. As shown in Figure 6, this approach is effective only up to some point. In our case, the appropriate value of should reduce the noise to a level that is acceptable for the algorithms we will use.

Secondly, we need to establish a reasonable game time limit – preferably the shortest time that will allow us to observe and evaluate each agent’s behaviour. At this point we can realize that if, for instance, we set , which in fact is quite low, and each evaluation’s time limit to 800 seconds, then the evaluation of 10 different configurations will require 50 evaluations of function. This will last more than 11 hours, which is far too slow when optimizing 11 continuous parameters.

Fortunately enough, the Quake II’s source code is available and was successfully altered in order to speed up the game. The in-game time calculation code and the client-server messages exchange rate have been changed. The size of the incoming messages buffer and the timeout values have been adjusted appropriately. Finally, the QASE API code had to be changed to be able to communicate with the modified Quake II server. The final effect is about 100 times game speed increase, which allows us to perform 50 evaluation of in less than 7 minutes.

### Choosing optimization parameters

In this subsection the parameters that will be constant throughout the whole optimization process will be chosen – the number of games played when averaging (), the game time limit and the maximal number of evaluations that each optimization algorithm will be allowed to perform.

First, the game time limit – we need to choose the minimal value that will allow us to differentiate configurations well. After many experiments, the time of 800 seconds has been chosen for each game. In this time, bots with exactly the same configurations playing against each other usually get the score of approximately 14 points each.

Choosing the parameter, we initially set and perform 100 evaluations of with time limit of 800 seconds and a constant configuration , which gives in total 5000 games. Due to Quake II game speed modification, this could be performed in less than 12 hours. Now, the average variance of measurements of performed during each of the 100 evaluations of can be observed for different (Figure 13).

variance-in-n.emf

Figure 13: The average variance of measurements of in the function of the number of repetitions . We can notice that this result is consistent with the theory described in Subsection 2.2.2.

It is also possible, that the noise is influenced by **.** Although, during the experiments this influence did not show to be very strong, it is safe to choose a little higher value of to leave a margin for the possibility of a greater variation.

The final value of that will be used throughout the following experiments is 20. Along with the game time limit of 800 seconds and Quake II game speeded up 100 times, we can estimate that to evaluate 100 configurations with we will need about 4 hours and 30 minutes.

In order to compare the results of the optimization algorithms in the following subsections, each algorithm will be run with a limit of 500 evaluations of .

### Localized random search

To use localized random search (RAND) we need to choose the distribution and variance of the perturbation . In our case, the Gaussian distribution should work well. The variance should be adjusted to how sensitive each of the elements of is. In our case simple tests have shown that parameters sensitivity strongly depends on their and other parameters’ current values; which might be not very surprising taking into account the way the agent makes navigation decisions. After a few tests using small number of evaluations, the variance of , same for each element of ,was chosen.

Another parameter that will be used in RAND is the acceptance threshold , as in Algorithm 2. It will be set using standard deviation of the results of , where is a constant configuration. In order to estimate , the results of the same experiment as for setting the value of are used. The average variance of 100 measurements of , using is around , therefore and finally the chosen value of will be: .

Results

The results of the RAND algorithm are presented in Figure 14. We can notice a high variance of the results, which can be caused by the random exploration of the search space. However, the average result, until around iteration 300 is increasing, which can be the effect of the localized nature of the search – the algorithm searches for the solutions that are close to the best currently known configuration.

fitnessInEvals1.emf

Figure 14: The results of each evaluation for localized random search. The highest result is 10.4.

### Hill-climbing

For the hill-climbing (HC) algorithm, we need to set the step size , as in (2.13) and, again, the acceptance threshold parameter . The setting of the acceptance threshold will be the same as in the case of localized random search described above. The step size will be also similar to the standard step size of the RAND algorithm. As we chose the variation of the elements of to be , the standard step size in the RAND algorithm will be equal to the standard deviation, that is . In case of the HC, as we search in a more organized manner, we may want the step size to be a little smaller, as in basic version of the HC algorithm it is constant. After few shorter experiments, the step size has been chosen.

Results

The HC algorithm, as we can see in Figure 15, stopped after 97 evaluations. This is because, according to Algorithm 3, hill-climbing should stop if it did not find any better solution in its vicinity.

fitnessInEvals-stop.emf

Figure 15: The results of each evaluation for hill-climbing algorithm. The highest result is 9.15.

In order to allow the algorithm to continue, the step size can be decreased instead of stopping the execution. The hill-climbing with step decreasing (HCSD) is proposed with the initial step size set to 0.4. When there will be no better solution in the vicinity, the HCSD algorithm will decrease the step size by 50%. The results of the HCSD algorithm are presented in Figure 16.

fitnessInEvals1.emf

Figure 16: The results of each evaluation for modified hill-climbing with step size decreasing (HCSD).  
The best result is 11.4.

We can see how the HCSD algorithm continues the search after iteration 97, where the basic version of HC algorithm stopped. Also we can notice how the variation of the results is getting lower due to the smaller step size.

### Finite difference stochastic approximation

For setting the parameters of the finite-difference stochastic approximation (FDSA) algorithm, the semiautomatic method described in Algorithm 6 will be used. After settings, it has been observed that the step size when performing gradient estimation was too high for our problem, therefore the parameter has been reduced. The final values of the parameters are: ,, , and .

Results

Figure 17 shows the results of the FDSA algorithm. The best and average result is increasing at approximately constant rate until iteration 300. We can also notice a little higher variation of the results at the beginning of the algorithm.

fitnessInEvals1.emf

Figure 17: The results of each evaluation for the FDSA algorithm. The best result is 16.45.

### Simulated annealing

The simulated annealing (SIMA) algorithm requires us to define the annealing schedule and the budget of function evaluations we are willing to allocate for each temperature. For the stochastic optimization, we also need to adjust the value of the acceptance threshold parameter .

We will use the same value of as we chose for other algorithms in the earlier paragraphs. Now, we can make a choice whether we want to be positive or negative. As hill-climbing and finite-difference stochastic approximation algorithms are local optimum searches, we prefer the SIMA algorithm to rather search for another global optimum, therefore we assume there are many global optimums and we will use a positive value of .

To adjust the annealing schedule and the budget for each temperature, a few possibilities were analyzed and tested. One of the requirements that has been established is for the annealing schedule to set such a temperature at iteration number 100, that the probability of choosing the worse solution for would be less than 0.70. In a similar manner, other points have been chosen reflecting the desired annealing schedule, which led to establishing the following parameters: , , and the function evaluations budget allocated for each temperature was set to 10.

Results

The results of the SIMA algorithm (Figure 18) have a higher variance and, similarly to localized random search, we can see that the average result is increasing, which can be the effect of choosing the new random configurations basing on the best currently known configuration.

fitnessInEvals1.emf

Figure 18: The results of each evaluation for the simulated annealing algorithm. The best result is 9.25.

### Algorithms comparison

The FDSA algorithm optimized LearnBot’s configuration best, reaching the result of 16.45, which is over 5 points better than the second result of HCSD. The difference between the results of algorithms on consecutive places is approximately 1 point. The comparison of the algorithms’ best results is illustrated in Figure 19.

algResults.emf

Figure 19: Comparison between best results of each algorithm.

We can also compare the moving average result of the last, say 40 evaluations of each algorithm – Figure 20. We can notice that the HCSD algorithm was leading until around the 125th evaluation. The global search algorithms RAND and SIMA, as they generate new solutions randomly in order to explore the search space more, have relatively lower levels of the average result than FDSA and HCSD.

algResults.emf

Figure 20: Comparison between moving averages of the results of 40 previous evaluations of each algorithm.

## Evaluation

The experiments that were performed after the optimization process are described in this section.

Basically, we want to compare the results before and after the optimization, so the first part of the evaluation will be the same as described in the initial experiment (Section 4.1), with the difference that instead of ReferenceBot, LearnBot with the best found configuration will be playing against EraserBot. The evaluation will be performed on the same map that was used in the initial experiment, which is different from the map used during the optimization process. This allows us to observe how the optimized configuration performs in the new environment.

Later, the values of the best solution found will be compared with the initial configuration of ReferenceBot. Finally, the simple experiment involving human players will be described.

Results

In the evaluation experiment, LearnBot uses the best configuration found during the optimization process – the result of the FDSA algorithm. LearnBot most of the time has a little higher score than EraserBot throughout the whole game (Figure 21). At the end, the final difference between bots’ scores is approximately .

hc-sd.emf

Figure 21: The average score of EraserBot and LearnBot in the evaluating experiments. LearnBot uses the best configuration found using the FDSA algorithm.

During the game, LearnBot, similarly to ReferenceBot during the initial experiments (Section 4.1), did not explore the world as much as EraserBot. It was, however, collecting more different items than ReferenceBot.

We can take a look at what item categories ReferenceBot and LearnBot with the configuration found using FDSA algorithm were collecting (Figure 22). We can see, that LearnBot was picking up weapons and armour more often than manually configured ReferenceBot.

Armour could be a significant factor in agent’s result improvement – the initial level of health is 100, whilst the initial level of armour is 0. When the bot is hurt by the enemy, first the armour points are decreased, and only after it reaches 0 level, the health points will start to decrease.

However, the Figure 22 does not specify under what circumstances the armour and weapons have been picked up – the configuration parameters are strongly interdependent, and simply setting the armour weight to some higher value does not increase agent’s effectiveness as the configuration found with the FDSA algorithm does.

Figure 22: The item pickups by their categories for ReferenceBot and LearnBot using the configuration found with the FDSA algorithm.

Figure 23 presents the actual values of the weights in analyzed configurations. We can notice, for instance, that while the FDSA has decreased the aggressiveness weight, the enemy cost weight has been also decreased. It means that the agent will less often engage the enemy directly, but at the same time it is more likely to go towards the enemy if there is some useful item it can pick up there.

Figure 23: The actual level of weights for the initial configuration used by ReferenceBot  
and the configuration found using FDSA algorithm.

### Human player study

To find out how the optimized bot performs against a human opponent, a small experiment has been performed. Two subjects played with LearnBot with the configuration optimized by FDSA algorithm for 30 minutes. Subject 1 has a relatively little and the Subject 2 has a rather moderate level of experience in playing Quake II, however both of them are quite familiar with all the aspects of the game, and they know the map which they play on quite well. The results of experiments for Subject 1 and 2 are presented in Figure 24 and Figure 25 respectively.

vs me.emf

Figure : The score in time of LearnBot using configuration found with FDSA algorithm vs. human Subject 1. The Subject 1 scored 11, whilst LearnBot scored 25.

vs k.emf

Figure : The score in time of LearnBot using configuration found with FDSA algorithm vs. human Subject 2. The Subject 2 scored 13, whilst LearnBot scored 19.

As we can see, LearnBot performs slightly better than human subjects. As expected, the Subject 2 has better result than Subject 1. We can also see, that LearnBot scored less in the duel with Subject 2, as it played against more demanding opponent.

The expert level players, who, for instance, participate in international Quake II competitions, are able to win against EraserBot, therefore it is quite likely they are also able to win the duel against LearnBot. However, LearnBot is able to win with moderately experienced human players.

## Summary

The initial experiments have shown that EraserBot performs better than ReferenceBot. Relatively low difference of score between bots suggests that ReferenceBot is a competitive opponent.

The optimization has improved the result of LearnBot playing against ReferenceBot on the map designed on purpose for the experiments. The best results were achieved using the finite-difference stochastic optimization and the hill-climbing algorithm with the step size decreasing modification.

In the evaluation experiments, LearnBot using the best configuration found played against EraserBot. Both bots performed on a very similar level, which shows that LearnBot indeed improved in comparison with ReferenceBot, even though the evaluation experiment was performed on a different, from the one used for optimization, map. Finally, a short human study was performed, in which LearnBot won the duels with two moderately experienced human players.

# Conclusions

This thesis demonstrates an example of an FPS game bot implemented using common game AI techniques and the navigation module created on purpose. In Section 1.6 we stated two main research questions that we tried to answer throughout this thesis:

* *What is the effect of different optimization algorithms on the performance of the FPS game bot?*
* *Is optimization an effective way to achieve a more intelligent behaviour of the FPS game bot?*

This chapter presents the conclusions and the short summary of the main contributions of this thesis.

It is extremely difficult to compare algorithms in stochastic environments objectively, as they are very sensitive to their initial configuration and require a tremendous number of function evaluations – each algorithm should be run several times in order to observe its average performance. Nonetheless, when our goal is to optimize our gain function effectively, it is a good idea to use a few different algorithms, as, depending on the problem, some of them may perform better.

In our case, although the game environment is stochastic, appropriate optimization algorithms helped to improve agent’s performance noticeably. The finite-difference stochastic optimization algorithm gave the best results in this case. Our optimized configuration’s result was higher by 5.83 points relative to the initial configuration’s result against a third-party agent, although it was optimized using a different map from the one that has been used for the initial and evaluation experiments.

Taking insight into the results of the algorithms (Figure 19), we can observe that global optimization algorithms in our case performed worse than local search. One of the reasons could be that global optimization algorithms require a greater number of evaluations, which in our case would be problematic, as 500 evaluations take approximately 22 hours and 30 minutes.

Answering the second research question – whether optimization is an effective way to achieve more intelligent behaviour of an FPS game bot – we should take into account not only the final results of the optimization, but also the effort spent on implementing and configuring them. The complexity level of the implementation of each algorithm and the testing environment is similar to the complexity of the agent’s logic itself. However, choosing appropriate initial optimization algorithms parameters can be quite time consuming.

In our case, the finite-difference stochastic optimization algorithm required the greatest amount of time spent on configuring it. Relatively easy to configure, while having a good result, was the hill-climbing algorithm with the step decreasing modification.

Configuring the bot manually is more difficult – the environment is stochastic, so in order to collect reliable observations, we need to perform numerous evaluations. At the same time the game speed cannot be too high, as manual configuration requires us to observe how the agent behaves in order to know which aspect of its behaviour requires a change. What is more, the parameters being configured are highly interdependent and the gain function’s sensitivity to each parameter differs.

Stochastic optimization methods are in our case a good alternative to optimize the agent’s behaviour effectively.

## Main contributions

The main ideas and developments in this thesis can be summarized as follows:

* Autonomous Quake II bot has been developed using the architecture proposed in [15] and the world knowledge representation based on a waypoint map.
* A fuzzy logic based solution is proposed to implement the bot’s navigation module with adjustable weights that can be used to modify the agent’s behaviour.
* An experimental framework is developed allowing to evaluate different configurations of the bot and perform their optimization. It allows one to increase the game speed, which helps to deal with the stochastic nature of the game environment.
* The optimization of the bot’s navigation module configuration using stochastic optimization algorithms was performed. The best results were achieved using the finite difference stochastic approximation algorithm and a modified version of the hill-climbing algorithm with decreasing step size.
* The optimized solution has been shown to perform better than the manually adjusted configuration and on a similar level as a popular third-party Quake II bot. A short human study was also performed, showing that the optimized solution performed slightly better than two test subjects.

## Further work

Further development of the solution proposed in this thesis could involve using some heuristics obtained from the game that would allow us to perform the search for optimal weights in a more informed and efficient manner, such as the item pick up statistics or the damage inflicted on the enemy.

Furthermore, it may be better to optimize for a given game map only, rather than searching for a universal solution. Following this thought, the ultimate solution could be to optimize the bot’s behaviour on-line, i.e., learn while the game is being played on a given map and against given opponent(s). This would allow our agent to intelligently adapt to changes of the environment. However, until now, this kind of behaviour has been developed only to a limited extent in simple, deterministic environments built on purpose, like in [32].

# Attachments

## Glossary

**Bot -** A habitual, shorter form of robot. In computer games the term bot referrs to a program that is performing some game actions as if it was another player playing the game.

**Deathmatch** - The basic game mode in Quake II. It is a game performed on a particular map, in which the goal of each player is to eliminate as many opponents as possible, while staying alive for as long as possible.

**Demo** - In Quake II, demo is a file, that can be created during any game of Quake II, containing the full record of player’s activity during the session. Essentially it is a copy of the network stream received during a game. It is often used to demonstrate player skills or as a proof of the result of a particular match or tournament.

**FPS** - First Person Shooter games - a popular computer games genre.

**Map** - In Quake II and many other computer games it refers to a particular environment in which the game is played. Typically, each game provides many different map.

**NPC** - Non-Player Character - characters appearing in the computer game that are controlled by the game program and not by human player.

**Respawn** - In Quake II it refers to a phenomenon of reappearing of the player in the game again, after he or she has been eliminated by the opponent. This can be seen as getting a "next life". The player appears at randomly chosen spawn point.

**Spawn point** - In Quake II, a point on the map at which players reappear or respawn to the game.

# Bibliography

[1] G. Luger, *Artificial intelligence : structures and strategies for complex problem solving*, 5. ed. Harlow England ;;New York: Addison-Wesley, 2005.

[2] A. Nareyek, “Computer Games - Boon or Bane for AI Research?.”

[3] “human intelligence,” *Encyclopedia Britannica*, 16-Jan-2011. [Online]. Available: http://www.britannica.com/EBchecked/topic/289766/human-intelligence.

[4] S. Russell, *Artificial intelligence : a modern approach*, 3. ed. Upper Saddle River N.J.: Prentice Hall, 2010.

[5] A. M. Turing, “I.—COMPUTING MACHINERY AND INTELLIGENCE,” *Mind*, no. 236, pp. 433-460, 1950.

[6] Encyclopaedia Britannica, inc., *The New Encyclopaedia Britannica.*, 15. ed. Chicago: Encyclopaedia Britannica, 2007.

[7] J. Gettler, “The First Video Game?.” [Online]. Available: http://www.bnl.gov/bnlweb/history/higinbotham.asp. [Accessed: 01-Aug-2010].

[8] Entertainment Software Association, *2009 Sales, demographic and usage data*. .

[9] P. Tozour, “The Evolution of Game AI,” in *AI Game Programming Wisdom*, 2002.

[10] J. van Waveren, “The Quake III Arena Bot,” Delft University of Tehchnology, 2001.

[11] J. Manslow, “Learning and Adaptation,” in *AI Game Programming Wisdom*, 2002.

[12] B. Gorman, M. Fredriksson, and M. Humphrys, “QUASE: An integrated API for imitation and general AI research in commercial computer games,” *Proceedings of CGAMES'05 conference*, 2005.

[13] P. Hingston, “A Turing Test for Computer Game Bots,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 3, pp. 169-186, 2009.

[14] “Botprize contest homepage.” [Online]. Available: http://botprize.org/index.html. [Accessed: 07-Aug-2010].

[15] P. Tozour, “First-Person Shooter AI Architecture,” in *AI Game Programming Wisdom*, 2002.

[16] R. Straatman, W. van der Sterren, and A. Beij, *Killzone's AI: dynamic procedural combat tactics*. Amsterdam: Guerilla Games, 2005.

[17] D. H. Hale, G. M. Youngblood, and N. S. Ketkar, “Using Intelligent Agents to Build Navigation Meshes,” 2010.

[18] P. Tozour, “Building a Near-Optimal Navigation Mesh,” in *AI Game Programming Wisdom*, 2002.

[19] C. Alavala and ebrary, Inc., *Fuzzy logic and neural networks basic concepts and application*. New Delhi :: New Age International (P) Ltd., Publishers,

[20] M. Zarozinski, “An Open-Source Fuzzy Logic Library,” in *AI Game Programming Wisdom*, 2002.

[21] M. DeLoura, *Game programming gems 2*. Hingham Mass.: Charles River Media, 2001.

[22] R. Yager, “An Approach to Inference in Approximate Reasoning,” *Intl. Journal of Man - Machine Studies*, no. 12, pp. 323-338, 1980.

[23] S. Rao, *Engineering optimization : theory and practice*. New York: Wiley, 1996.

[24] R. Bellman, *Dynamic programming*, 6. ed. Princeton N.J.: Univ. Pr., 1972.

[25] J. Spall, *Introduction to stochastic search and optimization : estimation, simulation, and control*. Hoboken N.J.: Wiley-Interscience, 2003.

[26] P. Laarhoven, *Simulated annealing : theory and applications*. Dordrecht ;;Boston ;Norwell MA U.S.A.: D. Reidel ;;Sold and distributed in the U.S.A. and Canada by Kluwer Academic Publishers, 1987.

[27] “Id Software: History.” [Online]. Available: http://www.idsoftware.com/business/history/. [Accessed: 19-Dec-2010].

[28] C. D. Harvey, F. Collman, D. A. Dombeck, and D. W. Tank, “Intracellular dynamics of hippocampal place cells during virtual navigation,” *Nature*, no. 461, pp. 941-946, Oct. 2009.

[29] “Internet Growth Statistics - Global Village Online.” [Online]. Available: http://www.internetworldstats.com/emarketing.htm. [Accessed: 08-Jan-2011].

[30] “Deathmatch - Quake II,” *Quake wiki*. [Online]. Available: http://www.quakewiki.net/archives/deathmatch101/q2/guide33bc.html?page=q2dm14.htm. [Accessed: 19-Dec-2010].

[31] H. Fuchs, Z. M. Kedem, and B. F. Naylor, “On Visible Surface Generation by A Priori Tree Structures,” *ACM Computer Graphics*, pp. 124-133, Jul. 1980.

[32] M. McPartland and M. Gallagher, “Learning to be a Bot: Reinforcement Learning in Shooter Games,” *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 78-83, 2008.

1. Bot - a habitual, shorter form of *robot*. In computer games the term bot refers to a program that is performing some game actions as if it was another player playing the game. [↑](#footnote-ref-2)
2. The inverse kinematics problem can be stated as a question: Given the desired position of the robot’s hand, what should be the angles at all robot’s joints? The forward kinematics problem seeks at what position will be robot’s hand providing the given angles at robot’s joints. [↑](#footnote-ref-3)
3. To be precise, there is no single, widely accepted simulated annealing algorithm. There are rather variations depending on the implementation details, all having in common the annealing principle. [↑](#footnote-ref-4)
4. Binary Space Partition Tree – a data structure for representing a virtual, three-dimensional environment, which can be used for an efficient collision detection and determining the order in which polygons should be rendered given the position of the viewer . [↑](#footnote-ref-5)
5. Original Quake II maps are usually suitable for games of up to 8 players. For games of two players only these maps may seem to be quite large. [↑](#footnote-ref-6)