Academic Year 2010/2011

Warsaw University of Technology

Faculty of Electronics and Information Technology

Institute of Computer Science



BACHELOR OF SCIENCE THESIS

Piotr Gwizdała

Optimizing the simulation of intelligent behaviour   
of a Non-Player Character   
in a First-Person Shooter computer game

Supervisor

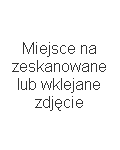
dr inż. Paweł Cichosz

Evaluation:

Signature of the Head

of Examination Committee

Information Systems Engineering

Date of Birth: 1987.06.02

Starting Date of Studies: 2006.10.01

Curriculum Vitae

I was born on July 2nd 1987 in Słupsk, the city located in the northern part of Poland. I attempted the 3rd High School in Gdynia, where I received my baccalaureate diploma in June 2006. Since October 2006 I study at Warsaw University of Technology at the faculty of Electronics and Information Technology. I work as a Java programmer since September 2009.

Signature of the Student

Bachelor of Science Examination

Examination was held on: 2011

With the result:

Final Result of the Studies:

Suggestions and Remarks of the B.Sc. Examination Committee:

SUMMARY

This work presents the autonomous agent designed to play a commercial First-Person Shooter computer game. The developed agent’s behaviour has been optimized using stochastic optimization algorithms and the final result has been evaluated against third-party game agent and human players, proving the effectiveness of the optimization and the competitiveness of the solution found.

In theoretical introduction the 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.

In the practical part, the developed solution’s algorithm has been described in detail. The optimization algorithms’ parameters as well as the optimization procedure were chosen and justified. Four optimization algorithms were applied and the results were compared. The final result has been evaluated against a popular third-party game agent and a simple human player study has been performed.

Keywords: artificial intelligence, computer games, stochastic optimization

Optymalizacja symulacji inteligentnego zachowania postaci   
w grze perspektywy pierwszoosobowej

Praca ta przedstawia autonomicznego agenta zaprojektowanego na potrzeby komercyjnej gry perspektywy pierwszoosobowej. Zachowanie stworzonego agenta zostało zoptymalizowane przy użyciu algorytmów stochastycznej optymalizacji, a wynik został oceniony poprzez porównanie z zewnętrznym agentem i z graczami ludzkimi, dowodząc skuteczności optymalizacji i konkurencyjności znalezionego rozwiązania.

We wprowadzeniu teoretycznym zarysowano powiązanie między dziedziną sztucznej inteligencji a komercyjnymi grami komputerowymi. Opisano metody sztucznej inteligencji wykorzystywane w grach perspektywy pierwszoosobowej oraz podstawowe metody optymalizacji stochastycznej.

W części praktycznej przedstawiono szczegółowy opis stworzonego rozwiązania. Parametry algorytmów i procedura optymalizacji zostały odpowiednio dobrane i uzasadnione. Zastosowano cztery algorytmy optymalizacji i porównano ich wyniki. Ostateczny rezultat porównano z zewnętrznym agentem i przeprowadzono prosty test z udziałem graczy ludzkich.

Słowa kluczowe: sztuczna inteligencja, gry komputerowe, optymalizacja stochastyczna

Contents

[1 Introduction 1](#_Toc283494197)

[1.1 Artificial Intelligence 1](#_Toc283494198)

[1.2 Computer Games and AI 2](#_Toc283494199)

[1.3 First Person Shooter games 3](#_Toc283494200)

[1.4 Motivation 4](#_Toc283494201)

[1.5 Thesis Overview 4](#_Toc283494202)

[2 Background 6](#_Toc283494203)

[2.1 Artificial Intelligence in First-Person Shooter games 6](#_Toc283494204)

[2.1.1 Bots architecture 6](#_Toc283494205)

[2.1.2 Navigation solutions 8](#_Toc283494206)

[2.1.3 Finite State Machines 11](#_Toc283494207)

[2.1.4 Fuzzy Logic 12](#_Toc283494208)

[2.2 Optimization methods 16](#_Toc283494209)

[2.2.1 Definition 16](#_Toc283494210)

[2.2.2 Stochastic optimization 17](#_Toc283494211)

[2.2.3 Direct random search 19](#_Toc283494212)

[2.2.4 Hill climbing 22](#_Toc283494213)

[2.2.5 Finite-difference stochastic approximation 23](#_Toc283494214)

[2.2.6 Simulated annealing 26](#_Toc283494215)

[2.2.7 Other techniques 29](#_Toc283494216)

[2.3 Quake II 29](#_Toc283494217)

[2.3.1 Quake II deathmatch 30](#_Toc283494218)

[2.3.2 Quake II client-server communication 31](#_Toc283494219)

[2.4 QASE API 31](#_Toc283494220)

[2.5 Eraser bot 32](#_Toc283494221)

[3 Developed solutions 34](#_Toc283494222)

[3.1 Creating a bot using QASE API 34](#_Toc283494223)

[3.2 Knowledge representation 34](#_Toc283494224)

[3.3 The main concept 36](#_Toc283494225)

[3.4 Navigation plan 37](#_Toc283494226)

[3.4.1 Criteria for choosing destination item 38](#_Toc283494227)

[3.4.2 Fuzzy logic application 41](#_Toc283494228)

[3.4.3 Enemy engaging plan 43](#_Toc283494229)

[3.5 Firing decision and instructions 44](#_Toc283494230)

[3.5.1 weapon and enemy choice 44](#_Toc283494231)

[3.5.2 Aiming algorithm 44](#_Toc283494232)

[3.6 Reference bot and Learn bot 46](#_Toc283494233)

[3.7 Bot’s launching and debugging application 47](#_Toc283494234)

[4 Experiments description 48](#_Toc283494235)

[4.1 Initial experiments 48](#_Toc283494236)

[4.2 Optimization process 49](#_Toc283494237)

[4.2.1 Results measurement 51](#_Toc283494238)

[4.2.2 Choosing optimization parameters 52](#_Toc283494239)

[4.2.3 Choosing Algorithms parameters 53](#_Toc283494240)

[4.3 Evaluation 55](#_Toc283494241)

[5 Results 56](#_Toc283494242)

[5.1 Initial experiments 56](#_Toc283494243)

[5.2 Optimization 57](#_Toc283494244)

[5.3 Algorithms comparison 61](#_Toc283494245)

[5.4 Evaluation 63](#_Toc283494246)

[5.4.1 Human player study 66](#_Toc283494247)

[6 Conclusions 68](#_Toc283494248)

[6.1 Main contributions 69](#_Toc283494249)

[6.2 Further work 70](#_Toc283494250)

[7 Attachments 71](#_Toc283494251)

[7.1 Glossary 71](#_Toc283494252)

[8 Bibliography 72](#_Toc283494253)

# Introduction

In modern Western World, the creation, distribution and manipulation of the information is becoming our main cultural and economical activity. We are transforming into *information society*. Many believe that this paradigm shift may have a similar impact on human lives as the Neolithic Revolution, when we transformed from hunting and gathering to agriculture-based way of living. Vivid visions of the future of information society invariably present our lives filled with computers, which are often more intelligent than most of humans. However, 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 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 now has become an inseparable part of our lives – the methods of artificial intelligence (AI) are used in weather forecasts, in digital photo cameras, internet search engines, business planning, medicine, and many others. One important field of application, and at the same time a good research platform, are commercial computer games [2], on which we will focus in this thesis.

In the first part of this chapter the core concepts used in this thesis are covered – the artificial intelligence, the computer games and their relation with AI, 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. The chapter ends with a brief description of the thesis goal and the outline of its contents.

## 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. However, there are many different 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 human brain, seeing AI as an empirical science, that involves hypothesis and experimental confirmation. On the other hand, others understand AI as a rather 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 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 if 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 were 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 state it using 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 not optimal.

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 considered 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 [5]. It is believed that the first electronic game was created by William A. Higinbotham – of the Brookhaven National Laboratory, in 1958, as interactive technology demonstration available for laboratory’s visitors. Tennis for Two was working on analogue computer and was simulating a tennis game on an oscilloscope. It was presenting 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 [6].

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 [7].

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 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 [8].

## 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 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 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

The 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 a realistic physics and vast interaction possibilities. However, in contrast 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. The 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 a research, as player’s actions usually have direct influence on the state of a game and his environment [9]. 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 among research community. For instance, in order to stimulate the research, Philip Hingston [10] proposes a variant of Turing Test designed for FPS bots on which the BotPrize competition is based. In the competition, taking place every year since 2008, the human players play with bots an FPS game, while being observed by judges. Basing only on observed behaviour of game characters, the judges have to tell the human players from bots. Till now, none of bots have managed to appear human-like enough to win the BotPrize [11].

## Thesis Overview

There are two main objectives of the practical part of this thesis:

1. Design and develop an autonomous FPS game agent that is able to compete with other players.
2. Improve the developed solution using optimization methods and compare it with third-party bots and human players.

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

Completing the second objective will give us better 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?*

# Background

In the first chapter we presented the core concepts of AI and computer games and showed what are the benefits of using the computer games in AI research. We introduced the FPS genre and argued why it is considered to be attractive from the researchers point of view.

In the beginning of this chapter, the basic methods used by developers of the AI in FPS games are presented. Later, we introduce some optimization algorithms, along with their stochastic equivalents. 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 the EraserBot – an example of Quake II bot.

## Artificial Intelligence in First-Person Shooter games

In this section the basic concepts and methods that are often used by game authors 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 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 [12] 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 Paul Tozour.   
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 are performing 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 chapter we will take a closer look at common game map representations. 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 combat, the combat controller should take over the control of most of bot’s behaviours, such as weapon and opponent selection, firing and manoeuvring or picking up items. The main challenge here is to quickly evaluate a situation and choose an appropriate tactic, which shows up 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 take 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, pre-defined by their authors behaviour. Another aspect of combat, that the combat component should control is the group tactics and communication between group members while in combat.

The behaviour component is one that controls all the other components and takes 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. As this is a managing controller, it’s quality will determine bot’s resulting behaviour.

### Navigation solutions

The 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. All that a bot needs to know is that there is a wall. All the 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 of chosen representation, most of modern computer games have it prepared by their creators (like in [13] and [14]) before the game is released. Although works like [15] 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.

The edges can be marked 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 an information that the given node is a good place to hide at.

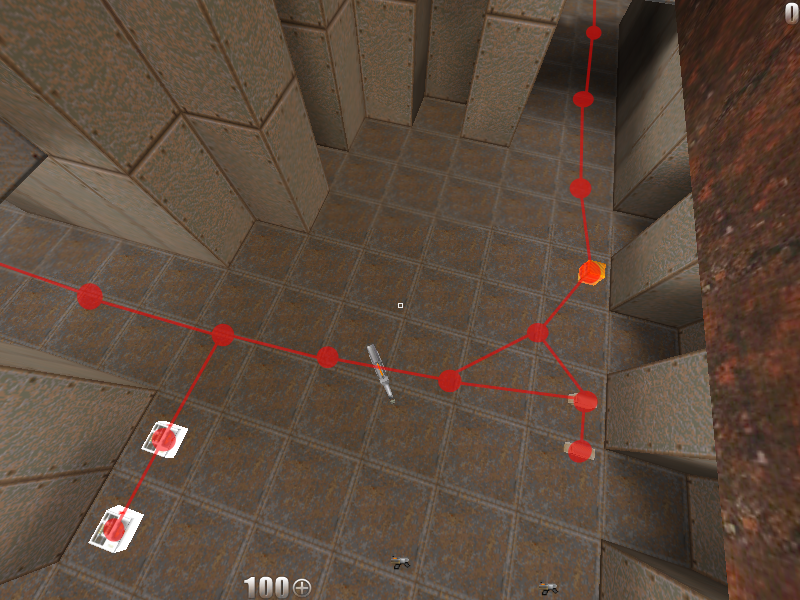


Figure 2: The example of the waypoint map.

It is important to make sure that moving from one node to another that is connected with an edge can be easily performed by a navigation module. Usually it means, that all 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 one of the 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 [15]. It divides all the walk-able surfaces of the environment into convex polygons, creating what can be called a “floor plan” of the world (Figure 3). Navigation mesh can also be represented as a graph in which nodes are polygons, and the graph edge exists between two nodes if their polygons sides overlap.

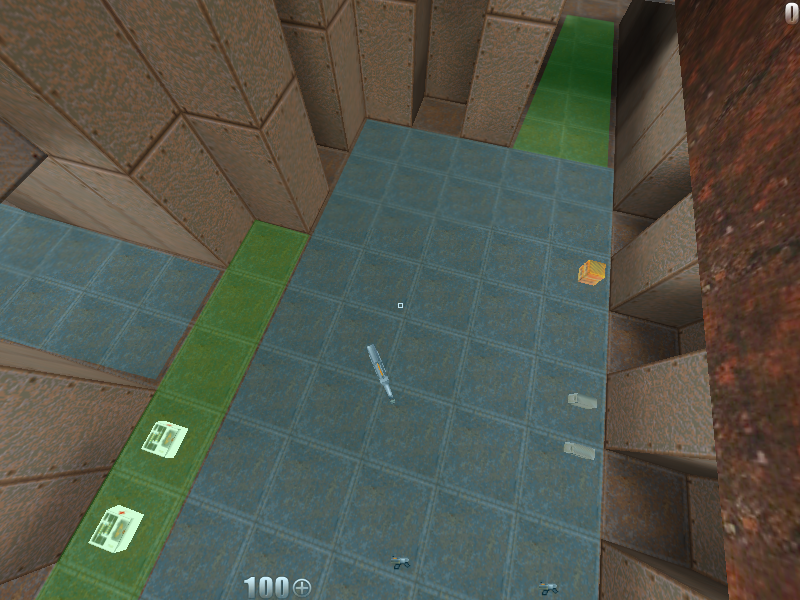


Figure 3: The example of the navigation mesh.

Navigation meshes are considered to be more powerful and providing more realistic navigation [16]. 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.

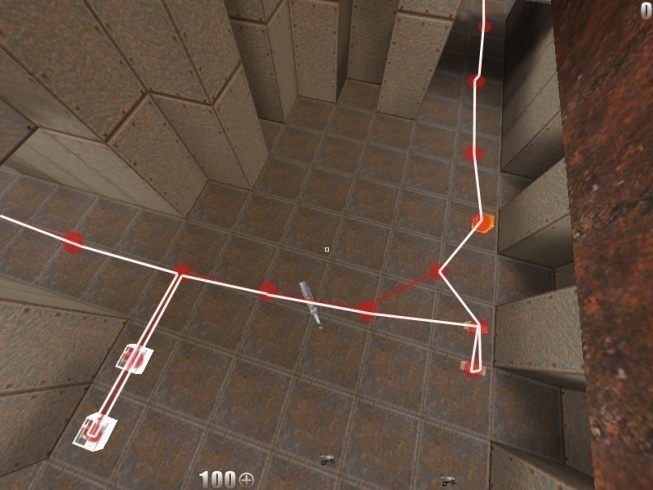


Figure 4: The comparison between the waypoint map and the navigation mesh. On the left picture the white line represents the path that the bot would follow in order to collect all the items in the room. On the right picture the bot performs the same action using the navigation mesh. The navigation mesh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

The difficult part in the navigation mesh is how a bot should move from one polygon to another. This 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. But if a human player will have enough time to take a closer look at our bot, the navigation mesh can often give 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 supported 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 on Figure 5. In this example, for instance, if a bot is in 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 trouble, manage a 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. 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 system [17]. 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 speed of calculation and ability to model complex behaviour [17] [18]. It is also often used in combination with other techniques, like presented in [19] 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 and expresses the degree of membership of an element in a fuzzy set . 1 is equivalent of 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 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 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 a range .

Example 2.

Let’s consider a binary fuzzy relation called “approximately equal” denoted and defined on set , where is a set of natural numbers. could be defined with a following membership function:

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

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. Membership function of Zadeh’s intersection of and is defined as follows:

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

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 fuzzy relation be defined as “bot needs item” and 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 a 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 a problem described in Example 2.2 is Ron Yager’s operators set [20].

Definition : Yager’s intersection and union operators

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

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

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 [17]. In the practical part of this thesis, the Yager’s operators will be preferred over Zadeh’s operators for a precise reason that was mentioned in Example 2.2.

## 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 obtain it [21]. In this section the principles of deterministic and stochastic optimization will be introduced. Later, the 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 the local and global optimization. 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 others are continues.

The difficulty of the optimization problem is closely related to the dimensionality of the vector . The volume of the search region grows geometrically 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 a “curse of dimensionality” – a term that was coined by Richard E. Bellman [22].

### Stochastic optimization

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

However, in real-life problems we often only have a noisy measure of the . This is when the stochastic optimization methods may be very helpful. In broader sense, the stochastic optimization problem can be defined as follows [23]:

Definition : Stochastic optimization

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

1. There is a 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 of the stochastic search algorithms that have the 2nd property may also be successfully applied when the gain function is deterministic in order to, for instance, speed-up the desired solution search process.

Problems that have the 1st property can be presented as optimization where the only information we have about the is a noisy defined as follows:

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

Where is a random noise, that can be dependent on the currently evaluated vector .

There are fundamental limitations of optimization with a noisy information about the gain function, as in (2.13). The statistical error of the gain function that will progress to the output of the optimization algorithm can only be reduced by significantly increasing the cost in 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. Figure 6 illustrates the relation between number of function measurement repetitions and the error . 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.   
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 from the evaluated vector , as even more gain function evaluations are usually required.

In the following chapters some basic, deterministic and stochastic optimization algorithms are outlined.

### Direct 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, the random search in some of the problems proves to be quite effective. Apart from that, these methods are often preferred over other algorithms because of the following reasons [23]:

* Random search is easy to implement – therefore the engineers can come about with a solution faster.
* Using exclusively the gain function – there are no additional requirements to the function, the gradient or hessian are not required.
* Reasonable efficiency – the random search algorithms often are able to provide reasonable solution quite quickly, especially when search is localized.
* Generality – the algorithms 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 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. This kind of search is often called a blind search and it belongs to the class of greedy algorithms.

The blind 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. The following algorithm, on the other hand, does use the currently best known solution . This property is sometimes referred to as a localized search. This term should not be confused with the local and global optimization, that were mentioned in subsection 2.2.1. In fact, some of the algorithms having a property of localized search, under certain conditions, are guaranteed to find the global solution.

Algorithm : Localized random search

1. Randomly choose the initial solution . Calculate the .
2. Randomly choose an independent vector , where is a 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 the vector will be chosen. The mean of the distribution should be zero and the standard deviation for each element of the vector should be consistent with the length of its possible values range. For instance, if the first element’s value is in a 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. Often it is beneficial to decrease the variance of , as the algorithm advances.

The main difference between localized random search and the blind search lays in the way the new random solution is generated. In blind search, the new solution is set with independently generated, new random vector. In localized random search, on the other hand, 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 most of the time search locations close to .

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

Noisy gain function case

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

First 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 on 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 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 the initial solution . Calculate the .
2. Randomly choose an independent vector , where is a 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 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, 2 times greater than , 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 a consequence of the noise. It is far more likely that the new solution is simply better than the actual.

Unfortunately, none of above approaches is perfect. The first one explicitly requires a significant increase in 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 is sometimes depending on the solution that is being 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 further improvement of given problem resolving process [23].

Hill climbing is, similarly to a basic random search method, a greedy search algorithm. Greedy algorithm is one that always chooses the best solution in the current situation, not caring about the consequences in a long run. In case of the hill climbing algorithm, its greedy approach in consequence may cause it to finish the search at the local maxima, therefore it is a local optimization algorithm, as it is explained in subsection 2.2.1. However, in many cases hill climbing can provide a good solution in shorter time than another 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 the solutions that differ with a current one in exactly one vector element, by either or , where is denoted step size, .

The actual hill climbing algorithm can be defined as follows:

Algorithm : Hill climbing

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

We can notice, that the algorithm performs an optimization 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 finds no improvement in the - it terminates. Another possible option would be to decrease the size of the step . This would cause the algorithm to search more carefully close to the location of the currently best known solution.

Noisy gain function case

When it comes to hill climbing algorithm and noisy gain function information, the 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.

Hill climbing, however, unlike localized random search, is a local search algorithm trying to rather 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, like it is in case of optimization with 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 function. 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 the initial solution . Let .
2. Let . Let .
3. If is sufficient or then stop and return . Otherwise go to step 2.

Where is a *step size*, that can be a constant, or an element of the predetermined sequence, or a solution to the secondary optimization problem: , called a *line search* [23].

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 modify each element of the vector in such way, that will move 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 given element, it means that the gain function is growing as the value of the given element increases. Therefore the element’s value will be increased.

Noisy gain function case

A stochastic version of Algorithm 4 exists, called *stochastic approximation* method, that can be used in case of the noisy gradient function. Many times, however, we do not even have the 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 yet effective method for gradient estimation is called a *finite-difference* method. Let’s denote the gradient estimation for step and 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 the when estimating its gradient. The finite-difference gradient estimation method can also be used in its one-sided form, when we sample just on the one side of the . In one-sided form, the -th row of the gradient estimation would be given with .

The following algorithm, called finite-difference stochastic approximation differs from Algorithm 4 only in 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 [23].

Algorithm : Finite-difference stochastic approximation

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

The parameters and from (2.16) are called gains of the algorithm. The theory of the convergence of finite-difference stochastic approximation algorithm to the local maximum expresses these parameters as follows:

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

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

Under certain conditions and appropriate choice of above parameters, the algorithm will approximately surely converge to local maximum. However, in practice we can often use less rigorous gain setting than the one required by convergence theory, which may prove to be more effective. In [23] the following approach is proposed to choose the values of the gain parameters:

Algorithm : Semiautomatic gain parameters setting method

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 in 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, that at each iteration tries to improve the current solution by adjusting every element of the vector using the gradient estimation. Hill climbing algorithm (section 2.2.4), on the other hand, adjusts just one element of at each iteration step. Finite-difference stochastic approximation method also reduces the step size at each iteration, allowing the search to focus on smaller area close to currently best known solution .

### Simulated annealing

In this subsection, the global optimization method basing on the annealing principle is outlined. Essentially, annealing algorithms are reducing the magnitude of random perturbations used in a 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 comes 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, that 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 a desired minimal energy state.

In analogy to optimization, the lowest energy state is the global optimum we are searching for, and the temperature is a 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 in 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 [24].

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 the energy state , and in a result of a random perturbation it is possible for a system to move to the 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 current system’s temperature.

Therefore, in 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 other algorithms described in this chapter), we can outline the algorithm called simulated annealing*[[3]](#footnote-4)*:

Algorithm : Simulated annealing

1. Randomly choose the initial solution .

Set an initial temperature . Calculate .

1. Randomly choose an independent vector , where is a dimensionality of . Let , make sure that .
2. Calculate . Let .
3. If , let . Else if and uniform on random variable 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.

We can notice, that at step 4, the Boltzmann constant was skipped, as we assume that we have a full control of 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 simulated annealing algorithm it is important to choose the appropriate cooling schedule, i.e. the sequence of values of the temperature. There are many possibilities, like decreasing 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 trial and error method.

Another important aspect that strongly influences the effectiveness of the algorithm is the distribution using which the 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 perturbation of each element of should be consistent with the range that a corresponding element of stays within.

Noisy gain function case

When the gain function contains a 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 a 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 the 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. Randomly choose the initial solution .

Set an initial temperature . Calculate .

1. Randomly choose an independent vector , where is a dimensionality of . Let , make sure that .
2. Calculate . Let .
3. If , let . Else if and uniform on random variable 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 in this case can be either positive or negative. When it is negative, then only new solutions that are having significantly higher noisy gain function value will satisfy the condition and will be chosen as a new best known solution. Otherwise, if , the worse solution will be chosen with less probability. The may be useful in case of gain functions in which we suspect the number of local maximums is low – we are less willing to give up our currently best known solution, unless the gain is big enough.

On the other hand, the 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 it’s noisy gain function value is even a little lower than the current solutions’ gain function value. Otherwise, the new solution still may 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 [25]. Four years after the first game version release, 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 [26].

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 [9]. Furthermore, we can directly observe interaction of our agent with human players during a game-play, which gives us opportunity to conduct experiments like BotPrize mentioned in section 1.4.

### 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 will die (and then immediately respawn back to game). The player dies when his or her health level drops to 0 or below (initially, after respawn being 100). During a 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 the enemies. There are 11 types weapon that differ with a damage they inflict, with the ammunition they require, the dispersion of projectiles, and gun reloading time.
* *Health* – used to recover from wounds by increasing the bot’s health level. There are different types of health items that provide different health benefit.
* *Armour* – reduces the damage a player receives from enemies. Generally, armour behaves similarly to additional health. When the player receives a damage from enemy gunfire, first his 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, different for each item threshold of time.

|  |  |
| --- | --- |
| 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 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 the game participants, executes them in its internal game environment representation and sends a world state update to each of the clients. Each of the clients 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 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 UDP network protocol. Thanks 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 [9].

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 section 2.1.2) – creating a waypoint map basing on pre-recorded data of 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 using a waypoint map only.
* Integration with MatLab® environment.

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

Bots developed using QASE API are client-side bots. As mentioned in subsection 2.3.2, Quake II server in order to keep all the clients up to date with the world information sends new messages to each client every 100 milliseconds. 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 access to full knowledge about the current game state. This includes positions of all the enemies and availability of items even in the most remote parts of the map, which gives a 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 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.

## Eraser bot

Since the Quake II game was first released in 1997, the number of internet users has grown over 28 times [27]. It is not surprising that many players who did not have constant access to 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 [28]: “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, the 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 teammates, like grouping in order to attack the enemy together.

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

# Developed solutions

In this chapter all the crucial decision making algorithms used by the agent that was developed are described. After short explanation of how the agents are created using 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 will not finish on time, the QASE will not send the new message to server and the server will assume that we repeat the actions we have performed in the previous frame.

The content of the runAI method will determine 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. However, implementing such an algorithm, as QASE API authors state themselves “is quite a challenge both from the perspective of autonomous agents and the perspective of artificial intelligence” [9].

## 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 skills or as a proof of the result of a particular match or tournament. However, we can also use a demo file to generate a waypoint map for our Quake II agent.

To do so, the waypoint map generator gathers all the positions in the world that have been occupied be the player recorded on given demo. Next, these positions are being 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 demo, creating a full waypoint map, that can be used by our agent. Because the geometry of the environment in Quake II doesn’t 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. In addition, each waypoint can store information about the item that can be found next to it.

Enemy information

Another important information, that the agent needs to store is the information about the opponents. Many times during the game, the enemy will disappear from our field of view. In this case it is more likely to find the enemy going towards the point where he 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 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 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 when was the last time when the given items was seen at its spawn position.

If the agent doesn’t see the item at the expected position, it can compute 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.

Another information that should be stored is whether given item is reachable for the agent or not. Not all the items that the agent can see it can actually pick up, 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 given item is reachable. To do so, we need to access the world geometry information stored in Binary Space Partition Tree provided by Quake II for each map and check if there is actually 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 effective players, besides having high firing accuracy during combat, are constantly picking up items that give them advantage over other players. 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 is chasing 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 agent’s combat module establishes firing decisions and shoots at visible enemies.

In this form, the algorithm constantly tries to improve agent’s inventory, health and armor state, at the same time taking every possible chance to inflict a damage on the enemy. The following is the basic explanation of the six most important steps of developed bot 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 a new information about the world given to the bot at 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 current bot’s state and situation in the environment. Then the path from current bot position to chosen item is obtained using the waypoint map.
3. Get navigation instructions – basing on currently executed navigation plan, the path that it provides and 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 – the bot basing on its own and enemies current position, the weapon the bot decided to use and its characteristics, it 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 paragraphs, the most important steps – 2, 4 and 5 are described in more detail.

## 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 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 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 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 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 a 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 exist so-called spontaneous plans. These plans are created when the agent is in the middle of 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 a 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 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. What does it mean that bot’s state is good enough to attack the opponent? This question will be answered after explaining how the agent chooses the destination item for a regular navigation plan.

### Criteria for choosing destination *item*

When 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. Below are explained all the factors that are used in the formula for calculating each item’s fuzzy value:

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 agent needs health and low, when agent has enough health.

Bot’s armour deficiency

Armour level value can range from 0 to 200. The armour deficiency is calculated similarly to health deficiency:

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

Where is agent’s armor level in state .

Bot’s weapon deficiency

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

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

Where is a set all weapons owned by a 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 exists 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, that 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.

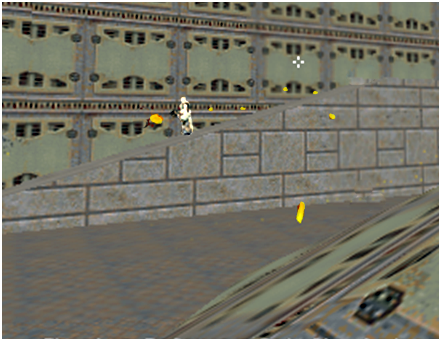


Figure 8: An example of *leading* technique. The agent aims at the point marked with a white crosshair. The yellow projectiles reach the enemy with a delay, when he will already change his position. Therefore, the agent needs to aim at the position, where it expects the opponent to be when the projectiles 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 on Figure 9.

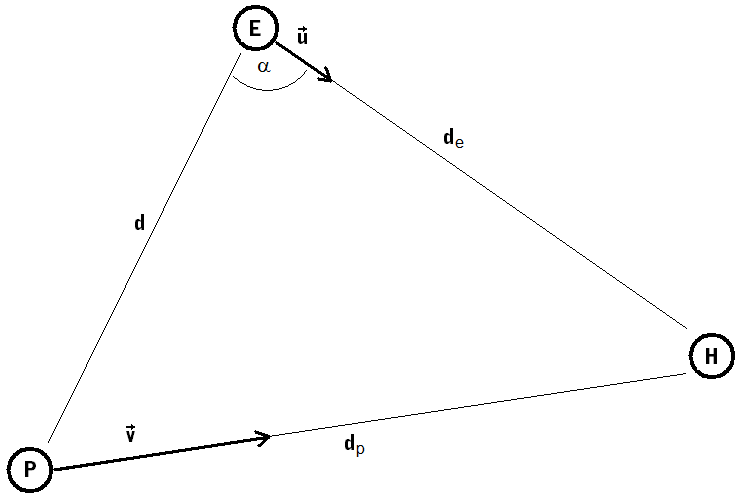


Figure 9: The illustration of the aiming task on a plane. Where is a position of the shooting agent, is a position of the enemy, is the enemies’ velocity and is velocity of a fired projectile, that we are seeking. If the enemy will continue to move with the velocity , the projectile will hit it at the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

## Reference bot and Learn bot

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 a LearnBot, whilst the agent whose weights were tuned manually by trial-and-error method and are constant throughout whole the thesis will be called a ReferenceBot.

## Bot’s launching and debugging application

* + 1. Launching
    2. Debugging
    3. Experiments ??
    4. Statistics

# Experiments description

In this chapter the experiments procedure is being described. There are three basic parts of the procedure: the initial experiments, the optimization and the evaluation of found solutions. In the following sections each of these parts is described and the parameters required to perform each step are chosen.

## Initial experiments

The aim of initial experiments is to obtain data that will allow us to compare the results before and after the optimization to observe the potential improvement.

The setting that will be evaluated during the initial experiments has been adjusted during the development process of the program. For instance, the agent often would be put in a situation when it has little health and little ammunition, and the ammunition items are close, whilst the health items are further. The weights were manually adjusted in such way, that the agent would rather choose to go for the health items, as the health level is more important. This method bases on authors intuition and has been evaluated mainly by the author himself. The resulting setting will be used as a setting for the ReferenceBot.

In the initial experiment, the ReferenceBot will be compared with the EraserBot. There will be played 5 separate deathmatch games, each of 30 minutes on the same map. One ReferenceBot will play against one EraserBot. The skill level of the EraserBot will be set to the highest possible. The same with the characteristics of the 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 will be probably the most popular Quake II map, called q2dm1, illustrated on Figure 10.

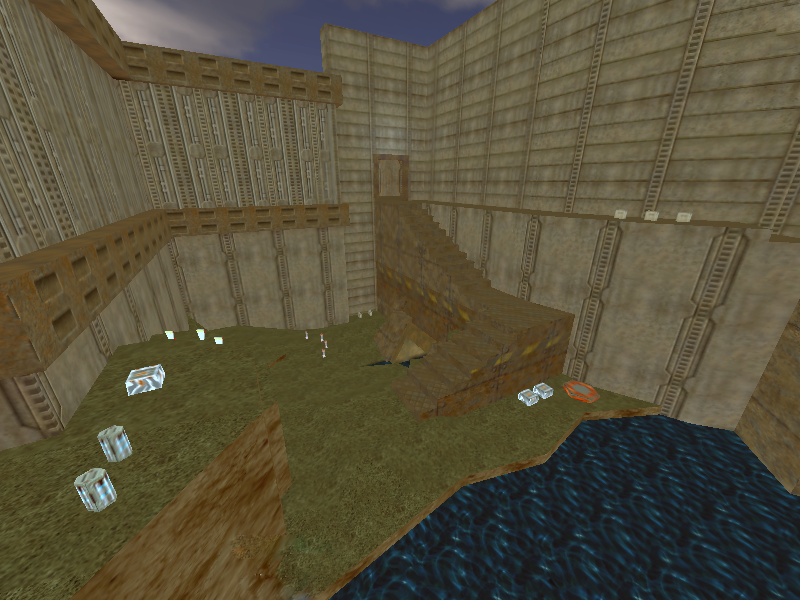


Figure 10: An overview of the part of *q2dm1* map, that will be used in initial experiments.

## Optimization process

This section describes the bot’s optimization procedure that has been applied. During the optimization, the LearnBot’s weight parameters used when establishing the navigation plan, as described in section 3.4, will be changed in such a way to possibly maximize bot’s result in the game against the ReferenceBot, which will be using 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 a LearnBot using configuration .

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

In order to make optimization more efficient, a designed on purpose map will be used. The map is smaller than standard Quake II maps, which will allow agents to play more dynamically[[4]](#footnote-5). It was designed to provide equal chances for each player, regardless where will he or she reappear after being defeated by the opponent. The relatively simple geometry of the map allows us to run the game at 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. The items on the custom map are distributed in a symmetric manner. The weapons with ammunition and health with armour are often separated, so the agent needs to make a good decision about what is more important for it. 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 on Figure 11.

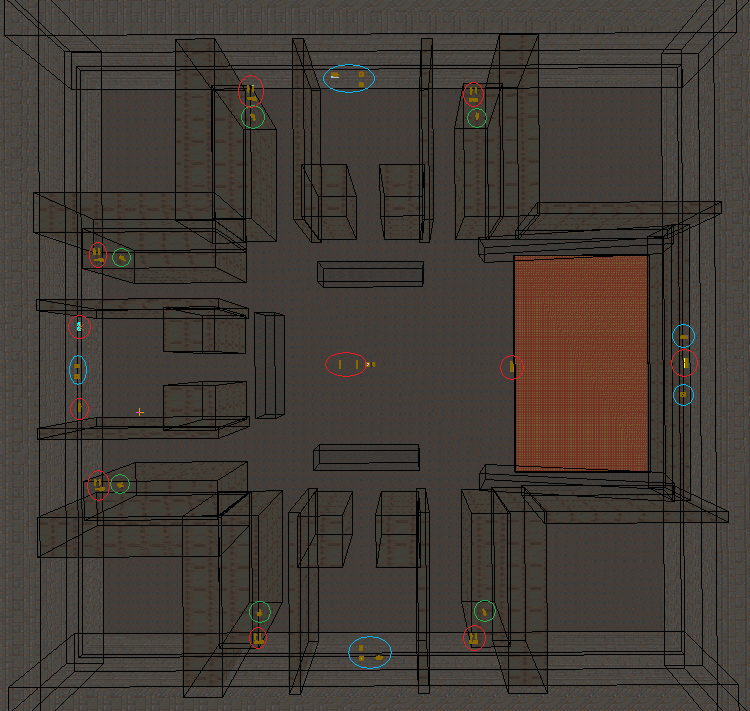


Figure 11: The overview of the map that will be used during the optimization process. The locations marked with green colour are player’s *spawn* or “reappearance” locations. The blue locations are those with health and armour items and the red ones are with weapons and ammunition.

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

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

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 assume, that regardless the algorithm chosen, each function evaluation will mean actually function evaluations and the average result will be treated as a final evaluation result. We can denote it as follows:

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

Although, we are decreasing the influence of the noise, it is not our goal here to remove most of it using just this method. As shown on 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 will be acceptable for the algorithms we will be using.

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. This has been observed to be about 10 minutes.

At this point we can realize that for instance, if we set , which in fact is quite low, and each evaluation’s time to 10 minutes, then evaluating 10 different configurations will last 8 hours and 20 minutes. This 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 were changed. The size of incoming messages buffer and the timeout values were adjusted appropriately. Finally, the QASE API code had to be changed to be able to communicate with modified Quake II server. The final effect is about 100 times increase of a game speed, which will allow us to perform the 10 evaluations mentioned in last paragraph in about 5 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 and the game time limit.

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 was chosen for each game. In this time, bots with exactly the same configurations playing against each other usually get score of approximately 14 points each.

Now, the parameter needs to be chosen. To do that, the was set initially to 50 and 100 evaluations of with time limit of 800 seconds were performed, which gives in total 5000 games. Thanks to the increased speed version of Quake II, this could be performed in less than 12 hours. Now, the average variance of measurements of performed during each of 100 measurements of in the function of value of can be observed (Figure 12).

variance-in-n.emf

Figure 12: The average variance of measurements of in the function of the number of repetitions . We can notice, that this result is consistent with theory described in subsection 2.2.2.

It is also possible, that the noise is influenced by **.** Although, during 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 possibility of a greater variation.

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

### Choosing Algorithms parameters

Localized random search

To use localized random search 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 showed that parameter’s sensitivity is strongly depending on their current and other parameters’ values, which is not very surprising taking into account the way that agent makes a navigation decision. As each element of stays in a range of , we can choose one variation for all of them. After few tests using small number of evaluations, the variation of equal to was chosen.

Another parameter that will be used in a localized random search is the acceptance threshold , as in Algorithm 2. It will be set using standard deviation of the results of . In order to estimate , the results of the same experiment as for setting the value of could be used, limiting to its established value of 20. The variance of 100 measurements of , using is around , therefore and finally the chosen value of will be: .

Hill climbing

In the hill climbing algorithm, we need to set the step size , as in (2.14) 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. When it comes to step size , we will also use the step size we desired to use for localized random search. As we chose the variation of the elements of to be , the standard step size will be equal to the standard deviation, that is . In case of a hill climbing, 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 hill climbing algorithm it is constant. After few shorter experiments, the step size was chosen.

Finite difference stochastic approximation

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

Simulated annealing

The simulated annealing 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 acceptance threshold parameter . This we have already adjusted 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 simulated annealing 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 one of the requirements that was established is for the annealing schedule to set such a temperature at iteration number 100, that the probability of choosing the worse solution in case of would be lower than 0.70. In a similar manner, another points were chosen, which led to establishing the following parameters: , , and the function evaluations budget allocated for each temperature was set to 10.

## Evaluation

In this section, the experiments that will be performed after the optimization process are described.

Basically, we will want to compare the results before and after the optimization, so the first part of the evaluation will be exactly the same as described in initial experiments section (4.1), with a difference that instead of the ReferenceBot, the LearnBot with the best configuration found will be playing against the EraserBot.

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

# Results

In this chapter, the results of the experiments described in Chapter 4 are presented. First, the initial experiments, then the results of optimization using each algorithm and the comparison between algorithms, and finally the evaluation of the optimized solution.

For convenience, in this chapter, the following abbreviations of algorithm names will be used: FDSA – Finite-Difference Stochastic Approximation, HC – Hill Climbing, RAND – localized random search and SIMA – Simulated Annealing.

## Initial experiments

In initial experiments, the EraserBot proves to be more effective – the average difference between bots is .

init2.emf

Figure 13: The average score of each bot in the initial experiment. The ReferenceBot uses   
the manually adjusted parameters.

During the game it has been observed, that the EraserBot would often win using one of the weapons that was very rarely used and picked up by ReferenceBot. The EraserBot also was exploring 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

Below are presented the results of optimization algorithms. Each algorithm was tested using the settings described in previous chapter, with evaluations number limit set to 500.

Localized random search

fitnessInEvals1.emf

Figure 14: The results in evaluations for localized random search. The highest result is 10.4.

We can notice a high variance of the results, which should allow the algorithm to explore the search space better. The average result is increasing until around iteration number 300, which can be the effect of localized nature of the algorithm – the new random configurations are chosen basing on the best currently known configuration.

Hill climbing

fitnessInEvals-stop.emf

Figure 15: The results in evaluations for hill climbing algorithm. The highest result is 9.15.

The HC algorithm, as we can see on 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.

In order to allow the algorithm to continue, the step size will be decreased instead of stopping the execution. The following results are for a modified version of the HC algorithm, where the step is decreased by half whenever there are no better solutions in vicinity. The initial step size in this case was set to .

Hill climbing with decreasing step size

fitnessInEvals1.emf

Figure 16: The results in evaluations for modified Hill Climbing with step size decreasing (HCSD).   
The best result is 11.4.

We can see how the Hill Climbing with step decreasing (HCSD) algorithm continues the search after iteration 97, where the version without the step decreasing modification stopped. Also we can notice how the variation of the results is getting lower due to smaller step size.

Finite difference stochastic approximation

fitnessInEvals1.emf

Figure 17: The results in evaluations for the finite-difference stochastic approximation algorithm.   
The best result is 16.45.

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 in the beginning of the algorithm.

Simulated annealing

fitnessInEvals1.emf

Figure 18: The results in evaluations for the simulated annealing algorithm. The best result is 9.25.

The results of the SIMA algorithm 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.

## Algorithms comparison

The FDSA algorithm optimized the 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 algorithms’ best results is illustrated on Figure 19.

algResults.emf

Figure 19: Comparison between best results of each algorithm in evaluations.

We can also compare the moving average result of last, say 40, evaluations of each algorithm – Figure 20. We can notice, that the HCSD algorithm was leading until around 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 average result than FDSA and HCSD.

algResults.emf

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

## Evaluation

In evaluation experiment, the LearnBot is using the best configuration found during the optimization process – the result of the FDSA algorithm. The LearnBot most of the time has a little higher score than the EraserBot throughout the whole game. 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. The LearnBot uses the best configuration found using the FDSA algorithm.

During the game, the LearnBot, similarly to ReferenceBot, did not explore the world as much as the EraserBot. It was, however, collecting more different items than the ReferenceBot.

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

Armour could be a significant factor in agent’s result improvement, as a bot can collect up to 200 units of armour, whilst just 100 units of health. Also, the initial level of health is 100, and 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 were picked up – the configuration parameters are strongly interdependent and simply setting the armour weight to some higher value did not increase agent’s effectiveness as the configuration found with FDSA algorithm does.

Figure 22: The item pickups by their categories for the ReferenceBot and the 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 FDSA decreased the aggressiveness weight, the enemy cost weight was 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 optimized bot performs against human opponent, a small experiment was performed. Two subjects played with LearnBot with the optimized by FDSA configuration for 30 minutes. Subject 1 has a relatively little and the Subject 2 has a rather moderate level of experience in playing Quake II game, however both of them are quite familiar with all the aspects of the game, and they know the map on which they play quite well. The results of experiments for Subject 1 and 2 are presented on Figure 24 and Figure 25 respectively.

vs me.emf

Figure : The score in time of the LearnBot FDSA vs. human Subject 1.   
The Subject 1 scored 11, whilst the LearnBot scored 25.

vs k.emf

Figure : The score in time of the LearnBot FDSA vs. human Subject 2.   
The Subject 2 scored 13, whilst the LearnBot scored 19.

As we can see, the LearnBot performs slightly better than human subjects. As expected, the Subject 2 had better result than Subject 1. We can also see, that the LearnBot scored less in the duel with Subject 2, as it was more difficult game.

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

# Conclusions

This thesis demonstrates the example of the FPS game bot implemented using common game AI techniques and the navigation module created on purpose. In section 1.5, we stated two main research questions that we tried to answer throughout this thesis. This chapter presents the conclusions and the short summary of main contributions of this thesis.

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

In our case, although the game environment is stochastic, the appropriate optimization algorithms helped to noticeably improve agent’s performance. 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 initial configuration’s result against the third-party agent, although it was optimized using different map from the one used for evaluations experiments.

Looking at the results of algorithms (Figure 19), we can observe, that the global optimization algorithms in our case performed worse than the local search. One of the reasons for this could be that the global optimization algorithms require 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 the 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 implementation of each algorithm and their testing environment is similar to the complexity of the agent’s logic itself. However, choosing appropriate initial parameters for the optimization algorithms 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 step decreasing modification.

Configuring the bot manually is more difficult – the environment is stochastic, so in order to make the reliable observation, we need to perform many 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 configured are highly interdependent and the gain function’s sensitivity to change of each parameter differs.

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

## Main contributions

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

* The autonomous Quake II bot was developed using the architecture proposed in [12] and the world knowledge representation based on a waypoint map.
* A fuzzy logic based solution is proposed to implement bot’s navigation module with adjustable weights that can be used to modify agent’s behaviour.
* The experiments framework is developed allowing to evaluate different configurations of the bot and perform their optimization. It allows to increase the speed of the game, 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 a finite difference stochastic approximation algorithm and the modified version of hill climbing algorithm with decreasing step size.
* 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 the 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 in a more informed and efficient manner, such as the item picking 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 bot’s behaviour on-line, i.e. learn while the single 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 [29].

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

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

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] 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.

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

[26] 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.

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

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

[29] 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. Original Quake II maps are usually suitable for games of up to 8 players. For two players only games these maps may seem to be quite large. [↑](#footnote-ref-5)