Contents

[1 Introduction 3](#_Toc280535707)

[1.1 Computer Games and Artificial Intelligence 3](#_Toc280535708)

[1.2 First Person Shooter games 3](#_Toc280535709)

[1.3 Machine Learning and Computer Games 4](#_Toc280535710)

[1.4 Thesis Overview 4](#_Toc280535711)

[2 Background 5](#_Toc280535712)

[2.1 Artificial Intelligence in First-Person Shooter games 5](#_Toc280535713)

[2.1.1 Introduction 5](#_Toc280535714)

[2.1.2 Bots architecture 6](#_Toc280535715)

[2.1.3 Navigation solutions 7](#_Toc280535716)

[2.1.4 Finite State Machines 8](#_Toc280535717)

[2.1.5 Fuzzy Logic 9](#_Toc280535718)

[2.1.6 Other techniques 11](#_Toc280535719)

[2.2 Machine Learning in computer games (Leave it? Remove it?) 11](#_Toc280535720)

[2.2.1 Introduction 11](#_Toc280535721)

[2.2.2 Online and offline learning 11](#_Toc280535722)

[2.2.3 Learning methods in FPS games 12](#_Toc280535723)

[2.3 Optimization in computer games 12](#_Toc280535724)

[2.4 Quake II and QASE API 12](#_Toc280535725)

[2.4.1 Quake II overview 12](#_Toc280535726)

[2.4.2 QASE API 13](#_Toc280535727)

[2.5 Related work 13](#_Toc280535728)

[3 Project requirements 14](#_Toc280535729)

[3.1 Project scope 14](#_Toc280535730)

[3.2 Assumptions 14](#_Toc280535731)

[3.2.1 QASE API 14](#_Toc280535732)

[3.2.2 Main focus 14](#_Toc280535733)

[3.3 Limitations 15](#_Toc280535734)

[4 Developed solutions 16](#_Toc280535735)

[4.1 Reference bot 16](#_Toc280535736)

[4.1.1 How does making a bot look like in QASE 16](#_Toc280535737)

[4.1.2 Knowledge representation 16](#_Toc280535738)

[4.1.3 The loop 16](#_Toc280535739)

[4.1.4 Navigation decision 16](#_Toc280535740)

[4.1.5 Firing decision 16](#_Toc280535741)

[4.2 Learn bot 16](#_Toc280535742)

[4.3 Bot’s launching and debugging application 16](#_Toc280535743)

[5 Experiments description 18](#_Toc280535744)

[5.1 Reinforcement learning ?? 18](#_Toc280535745)

[5.2 Navigation optimization 18](#_Toc280535746)

[5.3 Experiments application 18](#_Toc280535747)

[6 Results 19](#_Toc280535748)

[6.1 RL ? went bad? 19](#_Toc280535749)

[6.2 Algorithms comparison 19](#_Toc280535750)

[6.3 Performance of found solution 19](#_Toc280535751)

[7 Attachments 20](#_Toc280535752)

[8 Bibliography 21](#_Toc280535753)

# Global to do

* Upewnić się, że poprawnie używam terminu *Frame of execution* opisanego w QASE API, p. 4.1.
* Przebudować navigation plan i powywalac powtarzające się kawalki z opisow rownan.

# Introduction

## Computer Games and Artificial Intelligence

The term *computer games* refers to interactive games operated by computer circuitry. The platforms on which computer games are played include personal computers, arcade consoles, video consoles and handheld devices [1].

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 analog 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 analog controllers to adjust the trajectory of the ball and a button to hit it with an invisible racket [2].

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 [3].

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 behavior 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 behavior in a computer game [4].

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 [4].

## First Person Shooter games

Computer games are divided into relatively small sets 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, the player sees the three-dimensional world through the eyes of his or hers character – from the first person perspective. In most of those games, a player has to fight against other players and NPCs using special items and firearms while navigating through three-dimensional real-world like environment.

This genre is also considered to be attractive for a research, as NPC’s actions usually have direct influence on the state of a game and its environment. In a football simulation game, for instance, the team’s result depends not only on particular NPC’s actions, but also on actions of other team members. There are also many similarities between controlling NPC in FPS game and controlling a real-world mobile robot, like a problem of path-finding [5].

## Machine Learning and Computer Games

## Thesis Overview

# Background

In the first chapter we showed … importance of the topic.

In the beginning of this chapter we introduce AI in First-Person Shooter games, presenting also some AI methods which are commonly used to accomplish bot’s tasks. Following, we present the summary of the Machine Learning techniques used in computer games. Next section depicts the concept of Reinforcement Learning and introduces the Connectionist Q-Learning algorithm. Finally, the last section describes ID Software’s Quake II as an example of an FPS game and QASE API – both used in the practical part of this thesis.

## Artificial Intelligence in First-Person Shooter games

### Introduction

First-Person Shooter games (commonly abbreviated to FPS games), are one of the most popular genres of computer games. In FPS, human players use a mouse and a keyboard to control their virtual in-game character. The main input for a player is the first-person perspective view of the world displayed on the screen and sounds played in the game. 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 armor jackets. Each FPS game is different, but usually, player’s health is described with some number and, if player’s health is poor, it can be recovered with a medical kit. If a player wears an armor jacket, the damage taken from gunshots will be reduced.

The form of the 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. In a perfect case in a result of performing optimal actions, the player should win the game[[2]](#footnote-3). In practice, games also have some random factors, that make them less predictable and more difficult to control.

The best human players are still better than the best bots. Although computers have potential to be better at some actions, like aiming and shooting accurately, in most of complex FPS games less precise human players still manage to develop tactics which allow them to win. Philip Hingston [6] proposes a variant of Turing Test designed for FPS bots on which the *BotPrize* competition is based. In the competition taking place every year, the human players play with bots an FPS game, while being observed by judges. Basing only on observed behavior 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*, but also all of them have been losing the game with human players [7].

### Bots architecture

The list of basic actions, that a player needs to perform in each game can be used to develop a generic architecture of FPS AI. Paul Tozour [8] proposes an architecture divided into four main components: animation, movement, combat and behavior. Figure 1 presents a diagram representing 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[[3]](#footnote-4), 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 behaviors, such as weapon and opponent selection, firing and maneuvering 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 behavior. Another aspect of combat, that the combat component should control is the group tactics and communication between group members while in combat.

The behavior component is one that controls all the other components and takes high-level decisions about bot’s behavior. 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 behavior.

### 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 a 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 and ) before the game is released. Although works like 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.

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 [11]. It divides all the walk-able surfaces of the environment into convex polygons, creating what can be called a “floor plan” of the world. 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 have a common edge.

Navigation meshes are considered to be more powerful and providing more realistic navigation . 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. Since we still use a graph, the path finding can be performed in exactly the same way as in case of waypoint maps.

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 is a better choice.

### 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 behavior and decide whether to fight or retreat. This kind of behavior 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 2. 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 : 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 behavior 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 . 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 behavior [14]. It is also often used in combination with other techniques, like presented in 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.

Fuzzy sets

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 n-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 n-ary relation a membership function is defined as follows:

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

Whilst for a fuzzy n-ary relation a membership function can take all values from a range .

For example a binary relation called “approximately equal”, defined on set , where is a set of natural numbers, could be defined with a following membership function:

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

To use fuzzy relations in bot programming, we usually need also some operations on them. The most basic operations on fuzzy relations are intersection and union.

Intersection and union operations

Let and be binary relations defined on set . The intersection of and is defined as follows:

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

The union of and is defined as follows:

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

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 relation of an intersection . If, on the other hand we are interested in an item that is close or a needed item, we will use a union operation.

Exhaustive compilation of many other possible fuzzy operations as well as more advanced fuzzy reasoning study can be found in .

### Other techniques

The range of techniques used in game AI programming is very wide.

## Machine Learning in computer games (Leave it? Remove it?)

### Introduction

As mentioned in the introduction…

### Online and offline learning

From computer games perspective it is important to differentiate between offline and online machine learning. When we employ machine learning techniques, we eventually want our agent to perform some task using learned knowledge. If our agent is learning while performing the task, we say it is learning online. If it learns before or after, we call it an offline learning.

Offline learning has already been used in many computer games, especially some sort of optimization or learning algorithms have been used to tune agent’s parameters, like in case of . In most cases the learning was performed in game publisher’s studio as a part of development process. It is indeed a very useful technique when trying to solve problems that are too complex to solve them manually, like driving a car on a slippery surface in *Collin McRae Rally 2.0* .

On the other hand, there are very few game titles that were using online learning techniques and none of them became a bestseller . Online learning is far more attractive for players, as it allows a character to adapt to conditions that it’s developers did not anticipate. For instance, if player develops a good strategy in game and can win easily, the online learning agent should be able to adopt to new conditions and develop appropriate counter-strategy in order to continue being a challenging opponent. Such an adaptive agent is considered to be more intelligent. Then why it is not widely used?

Although attractive, online learning has disadvantages. The first major disadvantage is the necessity of evaluating the learned behaviors in game using specific measures. Many times it is problematic to find an appropriate measure that would reflect the goal of learning. Also to minimize the impact of random factors, each evaluation needs to be executed for longer time in order to be considered reliable. It is usually too long for a human player . Another disadvantage of online learning is the fact that an agent can learn a behavior that is not desired, but effective. It may, for instance find an error in game physics and explore it, making a game unfair and therefore not attractive for human players.

### Learning methods in FPS games

* Reinforcement?

## Optimization in computer games

* Introduction – what are optimization methods and where are they used
* Usages of optimization methods in games
* Categories and descriptions of basic methods that are used in games
  + Hill climbing
  + Simulated annealing
  + Beam search
  + Genetic algorithm

## Quake II and QASE API

### Quake II overview

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 . 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 .

The game itself has been designed to be a challenging entertainment for humans, therefore it requires a full spectrum of reactive, tactical and strategic behaviors from a player, which is one of the factors that make Quake II attractive for AI researchers . Furthermore, we can directly observe interaction of our agent with human players during a gameplay, which gives us opportunity to conduct experiments like *BotPrize* mentioned in chapter 2.1.1.

### 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 *entities* which are distributed on the *map*. There are four main categories of *entities* that a player can pick up and use:

* *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 entities* that provide different health benefit.
* *Armor* – reduces the damage a player receives from enemies. Generally, armor behaves similarly to additional health. When the player receives a damage from enemy gunfire, first his armor 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.

*Entities* also *respawn* after they have been picked up by a player, but not immediately – each *entity* reappears after some specific threshold of time.

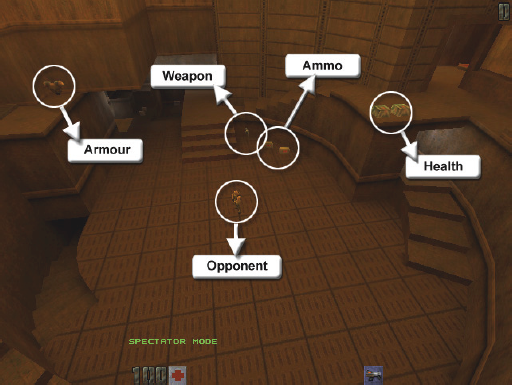


Figure : The Quake II game environment.

### 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 connecting, 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.

Since Quake II is a real-time game, the server sends updates of the world state to all its clients as often as every 100 milliseconds, using its own application layer protocol, basing on UDP network protocol.

### 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 .

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 behavior (actions)
* A set of tools for handling bot’s waypoint-based map representation (as described in section 2.1.3) – 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.

## Related work

* Use of QASE in other research and description of each one
  + Many of them were modifying existing bots
* Other research on Quake II + description

# Project requirements

## Project scope

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

1. Design and develop Quake II bot that is able to compete with other players.
2. Improve the developed solution using machine learning and optimization methods, comparing it with third-party bots and human players.

The first one will allow us to test in practice basic methods of FPS AI programming and will provide a base for further development.

Completing the second objective will let us evaluate suitability of methods used in this application as well as its implementing and testing complexity, which is of special interest from commercial game developers’ point of view.

Whether a bot is able to compete with human players and third-party bots will be verified with an experiment under conditions similar to a typical Quake II match.

## Assumptions

In this section basic assumptions of the practical part of the thesis are described.

A typical commercial bot AI is a complex program, for instance [9] or [10], and it is usually developed by a small group of people. This, to a degree, motivates the following assumptions, as having limited resources, the author had to focus on chosen, more specific aspects of bot development.

### Usage of QASE API

QASE API (described in section 2.4.4) allows development of Quake II bots focusing on AI rather than on game-specific issues like communication with a game server.

### Main focus

It is assumed that this thesis will focus on the combat skills of programmed bots.

In the early stage of the work, the author has noticed that it is relatively difficult to evaluate the quality of a particular bot’s navigation or general decision-making skills. At the same time the evaluation of bot’s combat skills can be quite direct, for instance, by measuring the inflicted damage to the opponent.

## Limitations

The main limitations of the project are determined by QASE API - all developed bots will be client-side bots.

As mentioned in section 2.4.3, 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. This creates a delay between bot’s perception and actions execution. For instance, if bot receives the information about enemy’s position at time , and decides to shoot at him, the shooting will take effect at time , at which the enemy may already be at different position.

On the other hand, a server-side bot may be programmed in such a way that it will react and perceive at the same, from other client’s point of view, time. This gives an important advantage to a server-side bot.

Another aspect of client-side bots is that the server informs them only about the world in their immediate surroundings, while 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 again, makes it easier for a server-side bot.

Although the development of a client-side bot has more constraints, it is worth noticing that from a human player’s point of view the client-side bot may be seen as a more fair opponent in comparison with a server-side one.

Another important limitation is regarding a comparison of developed solution with a third-party agents. QASE API is technically incompatible with some of popular Quake II bots that could be used for comparison. 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 works without any known issues with bots developed using QASE API.

# Developed solutions

## Introduction

In this chapter ...

## Reference bot

The Reference bot is a core part of this work. It has been developed from the scratch basing on the concepts referenced in chapter 2. As a name suggests, this bot will be used as a reference to compare it’s result with other agents.

## Creating a bot using QASE API

Using the features mentioned in section 2.4.4, 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 *runAI* 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. The content of this method will determine entire behavior 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” .

## 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. As waypoint map is rather static in Quake II, the geometry of the environment doesn’t change, the Floyd’s algorithm is used to compute the shortest paths between each pair of waypoints. In addition, each waypoint can store information about the *entity* 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 4.2.5.

Entities

Knowing that *entities*, 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 *entity* was seen at its *spawn* position.

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

Another information that should be stored is whether given *entity* is reachable for the agent or not. Not all the *entities* 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 entity 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 entity 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 Reference bot. At every moment of the game, the bot is either on its way to pick up some entity or it is chasing the enemy. This is done by establishing so-called *navigation plan* by choosing the *entity* 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 the Reference bot algorithm taken at each *execution frame*:

1. Update the knowledge base – this step updates the information described in section 4.2.2. 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 *entity* is chosen basing on current bot’s state and situation in the environment. Then the path from current bot position to chosen *entity* 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 – number 2, 4 and 5 are described in more detail.

## Navigation plan

The navigation plan essentially consists of a destination entity 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 we need to remember, that it is computationally expensive to establish a new plan at every execution frame. To accomplish 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 entity as a previous one. This is because the algorithm tries to choose the best available destination entity, and unless something important has changed in bot’s or environment’s state, the same destination entity 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 entity – 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 entity 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 entity for a regular navigation plan.

When the agent decides to create a new regular plan, it creates a ranking of all known entities that are possible to pick up, where each entity has assigned a real number that represents it’s rank – calculated using a heuristic described below. The entity with a highest rank is chosen. Below are listed factors that are used in the formula for calculating each entity’s rank value:

* Agent’s current state
  + 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 bot’s health level in state . The health deficiency is high, when agent needs health and low, when agent has enough health.

* + Bot’s armor deficiency – armor level value, similarly to health level can range from 0 to 100. The armor 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 efficient 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 of indexes of all weapons owned by a bot in state , is a set of indexes of all the weapons available in the game, and is a weight of a weapon with an index .

* + 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 for the guns with a great weight assigned. To calculate ammo deficiency the agent uses the following equation:

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

* Item’s characteristics
  + Item’s pickup benefit – each item influences one of the 4 characteristics of a bot state that were described above: health, armor, 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. In other words, how much would the deficiency in category decrease if the agent picked up the given item belonging to category :

Where is an item that we calculate pickup benefit for, is an item category to which belongs to, is current bot’s state and is bot’s projected state after picking up the item .

* + 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 to remain, 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 and is a shortest path from current bot’s position to item .

Having introduced all the above measures, we can finally calculate a rank value for each item we consider at bot’s state :

* + - 1. Sending movement to server

## Firing decision and instructions

### 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 efficiency 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 adjusted for each weapon. The weapon with the highest weight is always chosen.

The enemy choice has also proven not to be a 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 at, 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.

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 approach, taking into account just one position back. Using more than one position back resulted to be less efficient. Knowing the enemy position and the speed of the weapon projectile, we search for a vector - the direction of our shooting, as shown on Figure 4.

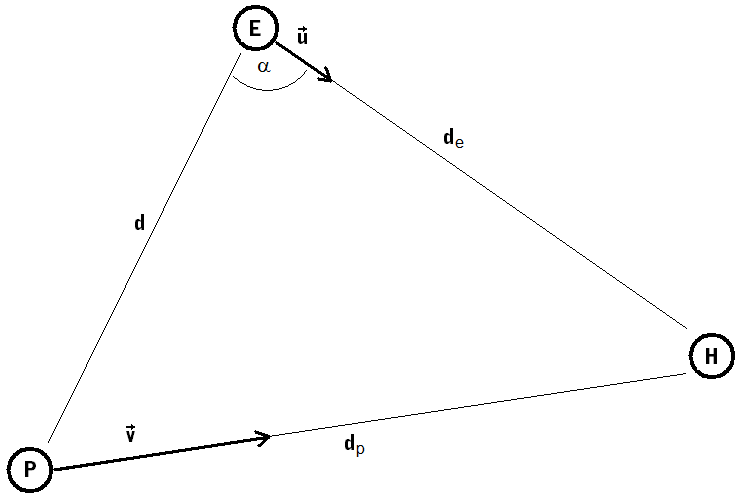


Figure : The illustration of the aiming task on a plane. Where is a position of the shooting agent, is a position of the enemy, is the enemies’ velocity and is velocity of a fired projectile, that we are seeking. If the enemy will continue to move with the velocity , the projectile will hit it at the 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 4: . Using this information and applying the law of cosines to the triangle from Figure 4, 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 we will be firing.

* + - 1. Sending firing instructions

## Learn bot

* + 1. Reference bot + changing the configuration

## Bot’s launching and debugging application

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

# Experiments description

## Reinforcement learning ??

## Navigation optimization

* + 1. Experiment desired characteristics
    2. Speeding up the game
    3. Choosing experiment procedure + justification

## Experiments application

* + 1. Algorithms chosen and why?
    2. GUI description

# Results

## RL ? went bad?

## Algorithms comparison

## Performance of found solution

* + 1. In-game observations
    2. With Eraser bot
    3. With human players

# Attachments

# Bibliography

[1] Encyclopaedia Britannica, inc., *The New Encyclopaedia Britannica.*, 15. wyd. Chicago: Encyclopaedia Britannica, 2007.

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

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

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

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

[6] P. Hingston, “A Turing Test for Computer Game Bots,” *IEEE Transactions on Computational Intelligence and AI in Games*, t. 1, nr. 3, s. 169-186, 2009.

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

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

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

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

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

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

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

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

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

[16] Generation5, “Interview with Jeff Hannan,” 2001. [Online]. Available: http://www.generation5.org/content/2001/hannan.asp. [Accessed: 19-Sierpień-2010].

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

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

[19] C. D. Harvey, F. Collman, D. A. Dombeck, i D. W. Tank, “Intracellular dynamics of hippocampal place cells during virtual navigation,” *Nature*, nr. 461, s. 941-946, Październik. 2009.

1. Bot - a habitual, shorter form of robot. The term bot is also used when referring to computer programs working on the Internet performing repetitive actions, such as price comparison, searching information etc. [↑](#footnote-ref-2)
2. Winning conditions of a game depend on a particular game scenario – it may mean, for instance, defeating all the enemies without hurting hostages. Depends on a particular game. [↑](#footnote-ref-3)
3. 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-4)