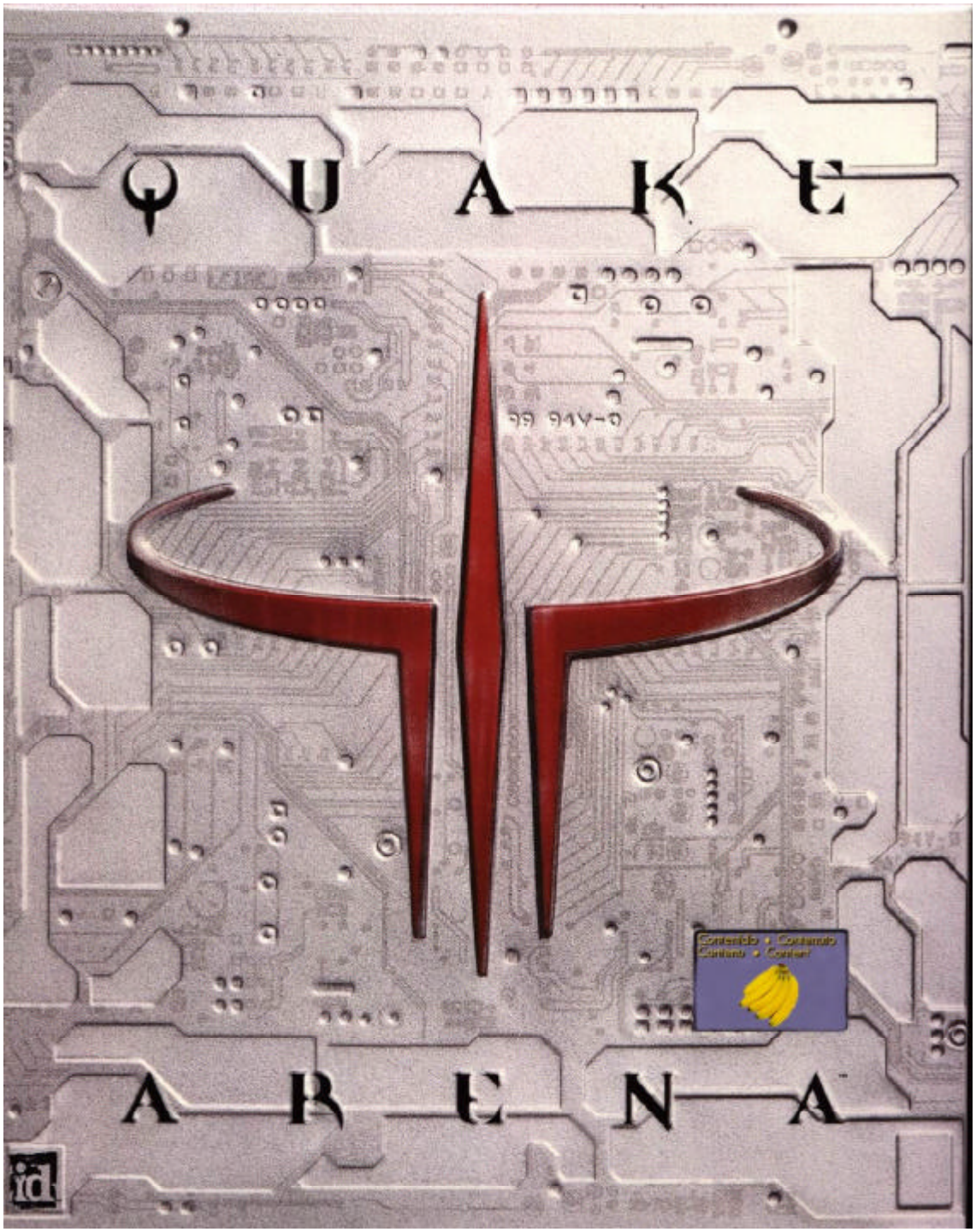

QUAKE
Contenido • Contenuto
Contenu • Content
ARENA
id

The Quake III Arena Bot

by J.M.P. van Waveren

June 28th 2001

Revision 1

University of Technology Delft
Faculty ITS

The Quake III Arena bot is an artificial player for the computer game Quake III Arena.

Abstract

Games play an important role in the field of artificial intelligence (AI). They offer an environment to test ideas about human reasoning, problem solving and other human abilities. The ultimate goal of AI is to create an artificial man. Games offer the opportunity to create artificial players which are modeled after human beings. Playing the game, human beings can interact with these artificial creatures and experience how well these creatures show human-like behaviour.

Quake III Arena is a computer game that belongs to the genre of first person shoot-em up games. The player views from a first person perspective and moves around in a real-time 3D virtual world. The most important tasks are staying alive and eliminating opponents within this virtual world. These opponents are other people, equal in strength and abilities, connected to the same game through a network or the Internet. The game has a set of different virtual environments called levels or maps, that contain rooms and hallways. The players have a whole range of weapons, items and powerups available to aid in the battles that take place in these maps. Quake III Arena offers various ways to play the game including team based gameplay modes.

This thesis presents the Quake III Arena bot which is an intelligent artificial player emulating a human player in the game environment. This artificial player is often called a bot as an abbreviation for the word robot. With this bot everyone can enjoy the game and practice, without the need for a network connection to other people. The bot is an artificial player that only 'lives' inside the computer, side by side with the game. The bot receives information about the game environment directly from the game program as a set of variables. The game input from the bot is also sent directly to the game program. Although the bot only 'lives' inside the computer, the ingame behaviour of the bot has to be hard to distinguish from the behaviour of human players. To make the game more enjoyable and more versatile, a range of different bot characters is used that each play the game in their own style, and provide different challenges for the human player.

To show human-like behaviour a wide range of techniques and common sense solutions are used for the bot AI. The Quake III Arena bot is the first commercially developed artificial player that uses fully automated path and route finding through arbitrary complex 3D polygonal worlds, without the need for the bot to acquire knowledge about routing and navigation during gameplay. No human intervention is required to provide the bot with all the information needed to navigate through, and understand new game environments. The bot uses a volume (area) based representation of the 3D game environment, which serves as a back-bone for the bot's cognitive world model. Together with a high performance path finding solution this cognitive model makes the bot rather resource efficient.

Key words and phrases: Quake III Arena bot, Quake3 bot, artificial player, artificial intelligence, automated route finding through polygonal worlds, navigation through polygonal worlds, hierarchical routing.

Acknowledgements

First of all, my thanks to id Software for making some of the best and most addictive games, and giving me the freedom to make the Quake III Arena bot reality as I deemed best.

Thanks to my supervisor and mentor at the University of Technology in Delft the Netherlands, drs. dr. L.J.M. Rothkrantz, for support, encouragement and inspiration.

It has always been a pleasure working with Miklos de Rijk. My thanks and appreciation to him for lots of input, feedback and bringing previously developed bots 'alive' with characters and ideas.

I would like to thank Alan [Strider] Kivlin for being a great sounding board and source of inspiration.

Special thanks to some of the best students at the university, Ronald Kroon, Edward van Bilderbeek and Karin de Boer. Both as friends and colleagues they have made life at the university a lot more enjoyable. Also my appreciation to them for keeping up with my mindless chatter.

Thanks to William van der Sterren for inspiration with his work on Computer Generated Forces (CGF) and for his many suggestions to this thesis.

Last but not least I am greatful to my family and friends for their love and encouragement.

J.M.P. van Waveren
June 28th 2001

Contents

List of figures

1. Introduction

1.1 Quake III Arena

Quake III Arena belongs to the genre of the first person shoot-em up games. A player views from a first person perspective and moves around in a real-time 3D virtual world. The most important tasks are staying alive and eliminating opponents within this virtual world. These opponents are other people, equal in strength and abilities, connected to the same game through a network or the Internet. The players have a whole range of weapons, items and power-ups available to aid in the battles. The game has a set of different virtual environments called levels or maps, that contain rooms and hallways. The battles in the game take place within these maps much like gladiators fight in an arena. Players can score points by taking out other players. When killed, a player respawns at one of the designated locations on the map and can continue to fight. Quake III Arena also has several team oriented gameplay modes. In normal teamplay there are two teams with players that fight each other. The team with the highest accumulated score, of all players on that team, wins. There is also a Capture The Flag (CTF) team based game mode. Again there are two teams with players. Each team has a base structure in the game world or map. A flag is positioned in such a base. A team scores points by capturing the flag of the opposing team and bringing it back to their own flag in their own base. More detailed information about the game can be found in appendix A. However playing Quake III Arena is probably the best way to acquire a better understanding of a lot of concepts in the game.

1.2 Artificial player

The Quake III Arena bot is supposed to act like a human player in the virtual world of the game. The bot replaces the need for other people to connect to the game. Just like a player can play the game with multiple other people the game can be played with one or more bots. To make the game more enjoyable and more versatile, a range of different bot characters is used that each play the game in their own style, and provide different challenges for the human player. In order to act like a human player the bot does not only need to understand the rules of the game and how the game works. The bot also needs basic abilities like navigating through the game environments, picking up items and handling weapons. Quake III Arena includes team based game modes like regular teamplay and Capture The Flag (CTF). The bot has to be able to play these game types and has to operate in teams. In order to operate in a team, with both human players and other bots, the bot needs to communicate with other players.
The bot lives inside the computer next to the game program and only appears as a human player in the game. The same (game) rules that apply to human players in the game, also apply to the bot. The bot does however not use the same input and output devices as human players. Instead of the output devices human players use, like the computer's monitor and sound card, the bot receives information about the virtual world directly from the game program as a set of variables. The bot also does not use the commonly used input devices like the keyboard and mouse. The bot sends a sequence of actions or intentions directly to the game program. These actions however, are very similar to the actions a human player can input using the computer's input devices. The

bot uses knowledge that has been provided in advance and knowledge acquired during gameplay to construct such sequences of actions.

1.3 Cognitive model

People often use different representations to deal with or remember different aspects of the environment they live in. Some people have a better visual memory. To remember things more easily they can try to visualize things that do not have a visual representation by default. Such different representations for different purposes are used a lot throughout life. In the same way the bot needs its own cognitive model of the virtual world it lives in. This internal model plays an important role in how the bot perceives and understands the virtual world. In particular, the bot cannot notice aspects of the world that are not represented within its cognitive model.

Figure 1.1: View in Quake III Arena.

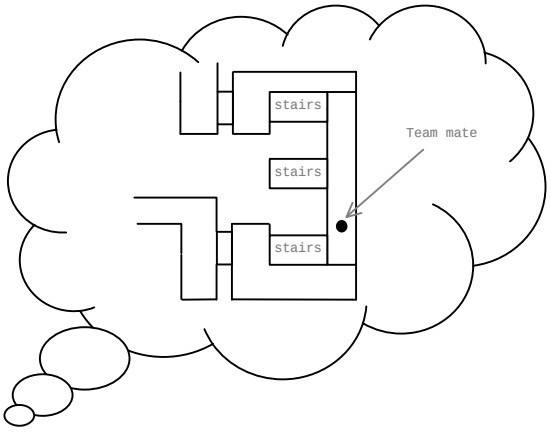


Figure 1.2: Cognitive model.

Usually the cognitive model used by a bot is a simplified version of the virtual world. If the bot were to live in the real world it would simply not be feasible to create a perfect model, because the real world is too complicated. Some of the models, available to model the real world, are quite sophisticated and accurate, but there are too many variables in play which, makes it infeasible to model the real world perfectly. Since the virtual world resides somewhere inside the computer, just like the bot itself, it is up to some degree possible for the bot to have a perfect model of this world. There can still be factors from outside the virtual world, for instance entities beyond the game program's control like human players, but most of the virtual world works according to explicit rules which can be modeled perfectly. However, a perfect model of the virtual world is usually not desired for several reasons. It would be rather computationally time consuming to use a model that simulates every single aspect of the virtual world. The required computation time grows very fast when the bot wants to evaluate a number of alternative courses of actions, and their influence on how the virtual world advances. Using a perfect model can also result in unrealistic behaviour of the bot. A bot would be able to look into the future and precisely predict what is going to happen. The bot would also be omniscient and could easily manipulate other entities in the world, and in effect set the world to its hand. However it is not desirable for the bot to have such an ability, not common to human beings. Aside from these problems, the representation used for simulating the virtual world is usually not suitable as a cognitive model for the bot. A different model is required, which allows the bot to perceive and understand aspects of the virtual world more easily.

For Quake III Arena the bot needs a cognitive model which allows the bot to represent the things required for autonomous behaviour within the virtual world. In it's simplest form this internal model can be a set of variables that represent the current state of the world. This state of the world includes the position of the bot, position of enemies and items, the weapons the bot has gathered etc. To be able to navigate through the virtual world and find certain locations in the virtual world the bot also needs a representation of the level or map the bot is situated in. Not all aspects of the 3D environment need to be included in such a representation. The bot will most likely care less about the textures and colors on the walls of the rooms the bot navigates through. Of course such colors might make opponents harder to spot, depending on their outfit, or perhaps have an influence on the mood of a player. However such influences are often not found to be significant enough to be taken into account.

1.4 Domain knowledge

For autonomous behaviour the bot also needs to maintain an explicit representation of how the virtual world changes. This knowledge is referred to as domain knowledge, and is required to reason about the effects of different sequences of actions. The reasoning is necessary for the bot in order to select actions, that are useful within the game and allow the bot to achieve certain goals. Human players often think intuitively about the effects of actions and make a lot of implicit common sense assumptions. A bot does of course not have this common sense by default. A certain level of domain knowledge and common sense will have to be built into the bot.
The Quake III Arena bot needs at least a basic level of domain knowledge in order to operate in the game. The bot needs to know about certain aspects of the virtual world that advance continuously as time passes. For instance when items are picked up by a player they will respawn within a certain amount of time. No effort from the bot is required for items to respawn, this happens automatically. Also some platforms in the environment continuously move up and down or back and forth. When the bot knows about the things that continuously advance or change, the bot can predict when certain items reappear, and the bot will know how long to wait before it can hop onto a platform in order to ride it.
A lot of things in the environment only change due to actions from players. Usually such actions only have an effect on other players, and little or no effect on the geometry of the environment. For instance when a player fires a weapon, another player might be hit and get damaged, which results in a loss of health for that player. However the walls, floors and ceilings of rooms in the environment show no or little change due to the impact of projectiles fired. Usually an impact marker is displayed where projectiles hit the wall, but the geometric representation of such a wall does not change. This means the bot cannot create shortcuts to other rooms by blowing up a wall. Sometimes actions do have an effect on the geometry in the environment. Some doors and bars are opened by pushing a button. In some cases the player has to walk up to such a button and press it, in other cases the player needs to shoot at a button in order to activate it.
How other players are likely to behave in reaction to certain events in the game world is also part of the domain knowledge. A player can perform certain actions in an attempt to influence other players. The player could walk into the direction of a certain item in order to make another player believe he is going for that item, but instead set a trap by waiting just around a corner. This kind of strategic domain knowledge is usually much harder to represent.

1.5 Knowledge acquisition

The bot does not only need basic knowledge in advance to be able to operate in the virtual world, the bot also has to acquire knowledge while playing the game. The current state of the world is one of the simplest kinds of knowledge the bot can acquire. This knowledge acquisition is referred to as sensing. Human players acquire knowledge about the current state of the world through the output devices of the computer. The player can look at the 3D image projected on the monitor and listen to sounds generated by the computer's sound card. To acquire more, or specific knowledge the player might need to look around or walk to a certain location for a better view or better acoustics. The bot however, does not necessarily need to perform special actions within the virtual world to acquire knowledge about the state of that world. The bot does also not use the same output devices used by human players. The bot receives information about the state of the world as a set of variables directly from the game program. Within the game program all the information about the current state of the virtual world is readily available. As a matter of fact there is more information available to the bot than a human player is able to acquire. The bot could peek at any information within the world since this virtual world resides inside the computer next to the bot. However it is usually not desirable for the bot to directly acquire information which is not available to a human player. For instance the bot should not be able to always know where it's opponents are within the virtual world. The bot is supposed to be a fair player and the ability to directly acquire more knowledge than a human player would be considered cheating. The bot may however predict where opponents are, based on knowledge that is available to any player. Of course the bot will also know the positions of opponents when they are visible to the bot. This however, requires a definition of what is visible to the bot and what is not. Based on a player's view direction only a limited portion of the surroundings are visible at any time. Certain aspects of the virtual world, for instance fog, can also blur the view or reduce the visibility of parts of the world. Such factors should also be taken into account when the bot acquires knowledge. The bot should not only be limited in the access to certain information in the virtual world, the bot should sometimes also acquire knowledge with a certain inaccuracy. When for instance several sounds are playing at the same time it can be hard to distinguish the different sounds. This can lead to an inaccuracy in the sounds the bot might recognize.
Aside from knowledge about the current state of the world the bot can also acquire knowledge about the dynamics of the world, the domain knowledge. The bot could acquire knowledge about how the world behaves and how other entities like other players within the world behave. This kind of knowledge acquisition is called learning and is far more complex than sensing. Often learning requires the recognition of patterns in a large amount of observations and evaluations of the effects of certain sequences of actions. Usually most, if not all domain knowledge a bot uses is provided in advance.

1.6 Bot behaviour

Just like a human player the bot has to be autonomous: the bot needs to decide how to behave on it's own. To be autonomous a computational model of the bot's behaviour is required. The bot's behaviour is defined by the sequence of actions it executes. Such sequences of actions could all be predefined. The bot would then always act the same and not adjust to different situation. The bots behaviour would be determined in advance. Predefined behaviour makes the bot rather predictable and will not make a very interesting opponent. Human players are often very unpredictable and a bot should have this same characteristic. Randomly choosing sequences of actions will make the

bot rather unpredictable, but this random behaviour is not particularly useful within the game. The bot has to choose its behaviour based on what it wants to achieve in the game. The domain knowledge and knowledge about the state of the world can be used to reason about and choose sequences of actions. The bot will want to choose only those sequences of actions that will lead to achieving the goals the bot sets out for. The bot has to choose these goals first and sometimes use several sub-goals in order to break down a complex task into several simpler tasks. In the teamplay game types the bot's goals might also be determined by a team leader who communicates certain tasks to the bot. During the course of the game the bot has to continuously evaluate if it comes closer to the goals it tries to achieve. The bot will have to adjust to changes in the game environment in order to achieve the goals.

1.7 Perfect simulation vs. enjoyable opponent

When creating an artificial player for a game, a natural approach is simulating a human player and how a human player thinks. One would try to simulate the human player into perfection. In order to simulate a human player the artificial player will need a certain degree of intelligence. As of now no-one has shown to be able to understand every aspect of human intelligence, let alone perfectly simulate a human being. However one can try to get as close as possible. This might seem feasible because within a game not all aspects of human intelligence have to be simulated, simply because not all aspects are applicable to the game. However a question arises. Is it really desirable to simulate a human player into perfection? And should an artificial player simulate exactly how a human player thinks and operates?
Another approach might also be plausible. One could try to create an artificial player that is not (easily) identified as being artificial and most of all is fun to play with. If an artificial player should not be easily identified as being artificial it still needs a certain degree of intelligence.

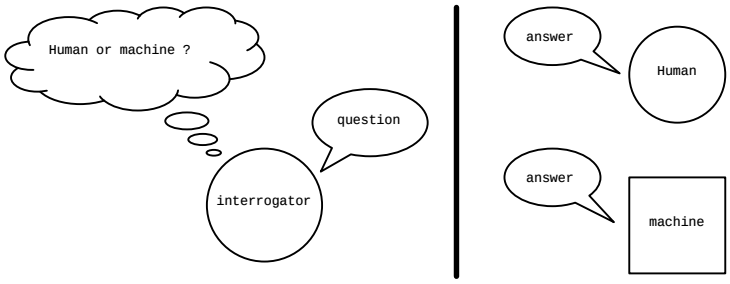


Figure 1.3: Turing test.

Alan M. Turing (1912-1954) came up with a test to identify machine intelligence. In 1950 he wrote an article about this test in "Mind, a quarterly review of psychology and philosophy" titled "Computing machinery and intelligence". In this test there are two people and the machine to be tested. One person and the machine are respondents, and the other person is an interrogator. The interrogator and respondents are all in different rooms. The interrogator can only ask questions via a keyboard or terminal. Both respondents attempt to convince the interrogator that they are the human respondent. The machine passes the test if the interrogator cannot tell the difference between the respondents, or guesses at chance at the identity of the respondents. If the interrogator can tell the difference the machine fails the test. Turing thought that any machine which passes the test should be considered intelligent or able to think. In other words, Turing

proposed the test as a sufficient criterion for machine intelligence. He felt it was not a necessary condition, because of the possibility that some intelligent creatures might not be able to correctly participate in the test for some physical reason. However, as Block (1995) shows it is possible to satisfy the Turing test with an unintelligent, physically possible machine. This means that the test does not seem to be a sufficient criterion either. If the test is neither necessary nor sufficient, perhaps it can be considered a mark of intelligence, rather than criterion for intelligence. The Turing test does provide the idea that for a machine to be considered intelligent, it is not required to operate in exactly the same way as existing intelligence. It is far more important that the machine appears intelligent and cannot be identified as being artificial.
Creating an artificial player that is not (easily) identified as being artificial and that is fun to play with seems a better approach, especially since the game Quake III Arena is meant to be fun and entertaining and the bots are part of the game. It is more important for people to have the illusion the bot is human and not artificial. However, one should also keep in mind that trying to more or less exactly simulate human intelligence and how humans think, is in some cases the best path to follow. In such cases it is the best way to make sure the artificial player is not identified as being artificial.

1.8 Generic vs. map specific knowledge

Even though the game play rules for Quake III Arena and the game itself seem not all that complex at first sight, there is a vast range of different and more or less complex strategies that can be applied. With the team based game modes included even more strategies can be thought of. A lot of strategy guides have been written for first person shoot-em up games. It is interesting to see that often only a small portion of such a guide discusses general strategies. The larger parts of a guide usually deal with level or map specific strategies. Usually the best strategies are listed per map. This is not without reason. Perhaps general strategies are harder to describe but they are always executed in different ways for each map or game environment. Since players usually play in a limited number of different maps, strategies can be described per map in these strategy guides.
Of course providing the bot with a certain degree of strategic knowledge is desirable. The above might lead to believe this knowledge needs to be implemented specific to a map or environment. To some degree the bot will need knowledge specific to a map. The bot will have to know it's way around in the 3D environments of the game. As will be shown, this type of knowledge can be deduced from the maps. The question is if strategic knowledge can also be deduced from the game environments by a bot. Human beings seem capable of doing this as they develop the map specific strategies. Is it also desirable to create a bot, which is capable of doing this? A lot of information can be deduced from the environment, like strategic positions, tactics [9] and perhaps even new strategies. One could also try to match a fixed set of strategies to a certain environment and try to find out which strategies are applicable. As to date this area of artificial intelligence in FPS games remains largely unexplored.
Providing the bot with all the strategic knowledge for each map by hand does not seem feasible. During the development of an FPS game the maps tend to change a lot. Designing and implementing a good map involves a lot of work. Implementing map specific strategies requires a lot of additional work and expertise. When the maps, and as a result the strategies change frequently, this is too time consuming and requires people with the right expertise.
However an approach somewhere in middle could be pursued. One could try to deduce as much information from the environment as possible. On top of this, scripting could be

used to aid the bot in certain situations. Such a script provides the bot with a specific plan, or tells the bot how to perform a specific task in its current environment. If a script resides at a relatively abstract level without referencing too much detailed information about a specific environment, then the script is also less sensitive to changes in a map.

1.9 Overview

The requirements for the Quake III Arena bot are outlined in section 2. Creating a bot for an FPS game is an extensive task which requires expertise from many different areas within the field of artificial intelligence. Some of the more commonly used methods and techniques will be loosely described in section 3. Relevant prior work related to both FPS games and the artificial intelligence in these games is reviewed in section 4. Section 5 describes how the Quake III Arena bot is structured. Section 6 through 17 describe the sub-systems that are used for the AI of the bot. The order of these sections is based on how the bot is built from the ground up. Section 18 provides some details on the implementation of the bot, and shows the results of some tests of the subsystems used by the bot. Section 19 concludes with the findings that surfaced during the development of the Quake III Arena bot. This section also provides some future directions for improvement and the development of new bots.

2. Requirements

The Quake III Arena bot has to act like a human player in the virtual world of the game. As such, the bot should be hard to distinguish from a human player. In particular the bot has to be visualized in the environment just like human players. The bot also has to navigate through the environment in a life-like manner, pick up items and handle weapons just like human players do. The bot has to be entertaining and suitable for practice and training purposes. To make the game more enjoyable and more versatile, there has to be a range of different bot characters that each play the game in their own style, and provide different challenges for the human player. The bot has to be a fair opponent and should in no event cheat. The communication with a bot should also be hard to distinguish from communication with human players. The bot has to be able to chat with other players. The bot also needs to communicate with team mates in the team based game modes like CTF. Life-like interaction with team mates is required to operate in a team with both human players and other bots.

The bot also has to meet a number of requirements on a technical level. The bot has to be resource efficient, both CPU and memory usage have to be low. Since the bot AI code is running concurrently with the game engine, the bot code may typically not consume more than 10 to 15% of the available CPU time. This is rather limiting especially since multiple bots will often play the game at the same time. The CPU usage should also be as constant as possible over time. Spikes in the CPU usage will cause interruptions in the game simulation, which are rather annoying for human players, and take away some of the gameplay experience. All the bots that exist in the game at the same time may not consume more than a few mega bytes (MB) of memory together. Any cognitive models and other kinds of knowledge, provided in advance or acquired during gameplay, have to fit in the limited amount of available memory.

The bot AI code has to be of commercial quality. Special attention has to be paid to the implementation and coding style. Both the architecture and code have to be robust and extendible. The code also has to be portable across various platforms because the game Quake III Arena ships for multiple different PC architectures and game consoles.
The Quake series of games are well known for their open structure, which allows third party developers to extend upon and modify the game. Third party developers have to be able to easily modify and customize the bot AI code to make the bots work with new game variants and modifications. Professional and amateur level designers can also create new game environments or maps for the game. The bot has to be able to understand and navigate through these new maps or environments without the need for complicated instruction from the level designers.

3. Background

Creating a bot touches many different areas within the field of Artificial Intelligence (AI), and also areas not directly associated with AI. This section by no means provides a complete and detailed description of the different algorithms and techniques used for the AI of a bot. However some of the commonly used methods and techniques are described.

3.1 Robots

An artificial player in an FPS game is often called a bot as an abbreviation for the word robot. The word 'robot' was coined by the Czech playwright Karel Capek from the Czech word for forced labor or serf. The use of the word Robot was introduced into his play R.U.R. (Rossum's Universal Robots) which opened in Prague in January 1921. The play was an enormous success and productions soon opened throughout Europe and the US. In part R.U.R's theme was the dehumanization of man in a technological civilization.
The term 'robotics' refers to the study and use of robots. The term was coined and first used by the Russian-born American scientist and writer Isaac Asimov (1920 - 1992). The word 'robotics' was first used in “Runaround”, a short story published in 1942. “I, Robot”, a collection of several of these stories, was published in 1950. Asimov also proposed his three "Laws of Robotics", and he later added a 'zeroth law'. 0: A robot may not injure humanity, or, through inaction, allow humanity to come to harm. 1: A robot may not injure a human being, or, through inaction, allow a human being to come to harm, unless this would violate a higher order law. 2: A robot must obey orders given it by human beings, except where such orders would conflict with a higher order law. 3: A robot must protect its own existence as long as such protection does not conflict with a higher order law.
According to these laws robots serve humanity in that they can perform certain tasks, and thereby take away the need for humans to perform these tasks. Bots in FPS games serve the purpose of entertainment and practice, and certainly do not always obey orders from human beings. However creating bots for an FPS game is closely related to robotics, especially the area of robotics that deals with life-like robots. Many of the same problems that arise in the field of robotics also surface when creating an artificial player for an FPS game. Both for robots and bots cognitive models play an important role in how the world they live in is perceived and understood. This world is usually the real world for a robot where a bot for an FPS games lives in a virtual world. However a lot of representations used to represent things in the real world can also be used to represent things in the virtual world. Both robots and bots in FPS games use sensing to acquire knowledge about the state of the world. Usually this sensing is far more complex for robots in the real world. Here sensing might involve recording images with a camera and recognition of patterns in such images. For bots in an FPS game retrieving the current state of the world is much easier. However the methods used for robot sensing often provide useful information on how to make a bot in an FPS game more life-like. Robots and bots can also both learn how the world advances under certain conditions or as a result of executed actions. The methods for acquiring and storing this kind of knowledge can be quite similar. Also navigating through the environment they live in is a complex problem for both bots and robots.

3.2 Path finding

One of the requirements for autonomous behaviour in FPS games is the ability to navigate around the game world in a life-like manner. Determining how to navigate through a map is an interesting problem. Many different approaches to solving this problem have been presented. Very simple AI just lets a bot walk forward until something is hit. At that point the bot turns and continues walking forward. There are also more complex path finding algorithms that use heuristics to find routes through the environment. A special representation of the environment or map is often used for these more complex algorithms. One of the most commonly used representations is a waypoint system. Such a waypoint system is a collection of points or locations (waypoints) with directional links between them. The waypoints represent the places where the bot can go and the links between them represent the paths the bot can follow in order to easily travel from one waypoint to another. Usually the links represent straight line directional paths. Creating an efficient waypoint system for a specific environment is an interesting and often complex task. A bot could create the waypoint system during gameplay and drop waypoints as it wanders through the environment. The bot will have a hard time reaching most places in the environment until it has wandered through most of the game world. Using this method the bot often never finds out how to go to certain hard to reach places. The waypoint system could also be created in advance before the bot enters the game. Placing waypoints throughout the environment and linking them is often a time consuming task to be completed by the level designer. Some algorithms have been developed to aid in creating a waypoint system in advance, but usually human intervention is required to optimize the system.

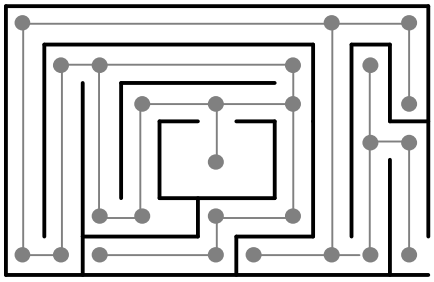


Figure 3.1: Maze with waypoints represented by dots.

When a good waypoint system is available to the bot, a whole range of different paths can be calculated. Usually only the shortest paths towards specific goals are used by a bot. However different kinds of paths can be useful as well. For instance paths that lead towards a goal, while avoiding certain areas of the world at the same time. Several algorithms are available to calculate the shortest path between a source location and a destination. The most commonly used algorithms are Floyd's, Dijkstra's and A* (A - star) [11]. These algorithms were designed in the context of graphs and graph theory. Since a waypoint system is very similar to a directed graph these algorithms can also be used to calculate paths along one or more waypoints. Traveling along a path, the bot might still encounter small or larger obstacles. A bot often uses sensing and environment sampling to identify the nature of such obstacles. The bot then tries to avoid or navigate around the obstacles. The Quake III Arena bot does not use waypoints to find routes and navigate through the environment. The path finding and navigation used by the Quake III Arena bot is described in section 6 and section 12.

3.3 Finite state machine

A finite state machine (FSM) is a system that has a limited number of states of operation. A real world example could be a light switch which is either on or off. The finite state machine that represents a light switch has an ‘on state’ and an ‘off state’.

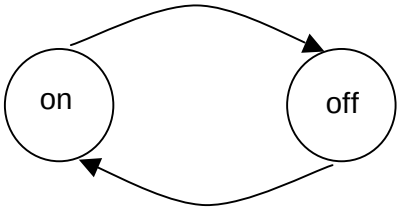


Figure 3.2: FSM for a light switch

In the light switch example there are only two states and from either state the light switch can change to the other state. Usually there are more than just two states and the state transitions are often limited. The subject being modeled usually cannot directly change from any state to every other state. The state transitions are also bound by certain conditions. The light switch for instance only changes state when a person pushes the switch in a certain direction.
Any system that has a limited number of possible states can be modeled as a finite state machine. Finite state machines are often used to simulate human beings, how they behave and think. Although there are other systems that can more accurately model the way humans think and learn, the simplicity of finite state machines makes them rather popular. The finite state machine only needs to be as complex as the desired complexity of the subject being modeled.

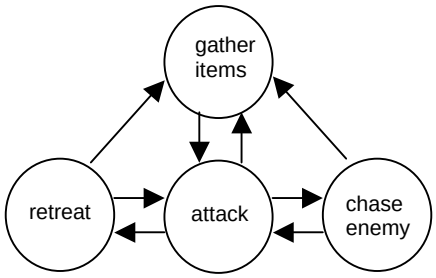


Figure 3.3: Simple FSM for a bot

Finite state machines are often used to model the line of thinking for a bot. The different states of the finite state machine can represent different states of mind, or different kinds of behaviour. There can be states for different situations and state transitions are often based on certain events in the game environment. A very simple bot could be modeled with a finite state machine using four states as shown in figure 3.3. The Quake III Arena bot uses a similar structure as the FSM to model its think process. This structure is described in section 15.

3.4 Fuzzy logic

Fuzzy logic [2] is a superset of conventional (Boolean) logic. This logic was extended to handle the concept of partial truth, also using values between "completely true" and "completely false". Dr. Lotfi Zadeh of UC/Berkeley introduced fuzzy logic in the 1960's as a means to model the uncertainty of natural language.

Fuzzy Subsets

Just as there is a strong relationship between Boolean logic and the concept of a subset, there is a similar strong relationship between fuzzy logic and fuzzy subset theory. Let's assume there's a set S and to all its elements there's one element of the set {0,1} attached. The subset U of the set S is defined as all the elements of S that have a '1' attached. The truth or falsity of the statement "x is in U" can be determined. The statement is true if there's a '1' attached to the element 'x' in S. Otherwise the statement is false.
Similarly for the fuzzy case there's a set S. But now a value from the interval [0, 1] is attached to every element of S. The subset U of the set S isn't strictly defined in this case. However it can be determined how much an element from the set S belongs to the fuzzy subset U. A value of zero attached to an element from S represents complete non-membership of U. A value of one represents complete membership. The values between zero and one represent intermediate degrees of membership. The degree to which the statement "x is in U" is true can also be determined. The degree of truth of the statement is given by the attached value to the element x in the set S.

Fuzzy functions and relations

Often a value from the interval [0, 1] is attached to an element of the set S using a function, the membership function. Such a function is one-dimensional because it's based solely on one criterion. In practice membership functions are based on two or even more criteria. Such a function gives a value from the interval [0, 1] to a combination of criteria and is often referred to as a "fuzzy relation". The criteria don't have to be elements from the same set. They can just as well be elements from different sets. The criteria also do not have to be elements from sets. Variables of some kind can also be used as criteria.

Fuzzy logic and bots

Fuzzy logic can be used by bots to express how much they want to have, or do certain things. For instance a bot might think of how much it wants to retrieve a certain item as a fuzzy value. The more the bot wants the item the higher the fuzzy value. Fuzzy relations can be used to express the relation between the current state of the bot and how much the bot wants something. For instance based on which weapon the bot is holding, and how much ammunition the bot already has for the weapon, the bot can attach a fuzzy value to how much it wants to retrieve more ammunition for the weapon. The Quake III Arena bot uses this kind of fuzzy logic to express how much it wants to have or do certain things. How the bot represents and uses the fuzzy logic is described in section 9.

3.5 Neural networks

A neural network (NN) [2] is a parallel distributed information processing structure based on the way biological nervous systems, such as the human brain, process information. This information processing structure consists of processing elements that are interconnected via signal channels, much like there are synaptic connections between the neurons in the human brain. Neural networks can differ in their number and arrangement of neurons, the way their neurons are connected, the specific kinds of computations their neurons perform and the way they transmit patterns of activity throughout the network. A human-like technique of learning is used to resolve problems with a neural network. Just as in biological systems, learning involves adjustments to the synaptic connections that exist between the neurons. A learning process called training is used to configure the neural network for a specific application, for instance data classification or pattern recognition. Different training strategies can be applied with different results and various learning rates.

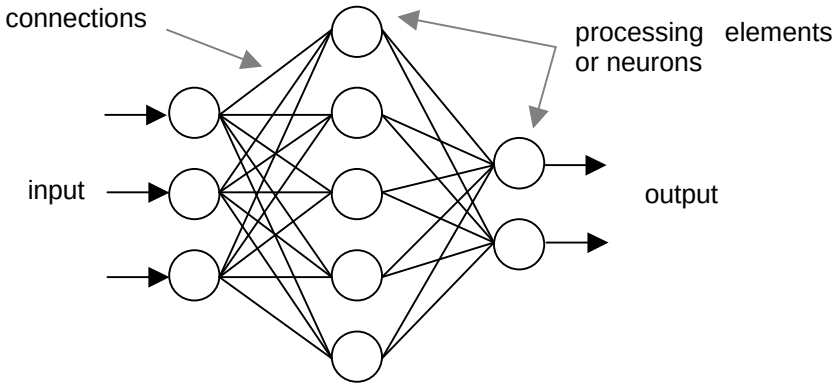


Figure 3.4: Example of a neural network

Neural networks are being applied to an increasing number of real world problems. Their primary advantage is that they can often solve problems that are too complex for conventional technologies, problems that do not have an algorithmic solution or for which an algorithmic solution is too complex to be defined. In general, neural networks are well suited to problems that people are good at solving, but that are hard to solve with programmed computing. These problems include pattern recognition and forecasting, which requires the recognition of trends in data.
Neural networks can also be used by bots to store and acquire different kinds of knowledge. Training of neural networks can be especially useful for a bot to acquire certain kinds of domain knowledge. Much like fuzzy logic a neural network can also be used by bots to express how much they want to have, or do certain things. A neural network can be trained in advance before a bot enters the game. During the game the bot will then only retrieve knowledge stored in the neural network. It is also possible to train a neural network during gameplay, which allows the bot to acquire all kinds of knowledge. However training of a neural network is often a time consuming process. The learning capabilities of the neural network will often have to be limited due to the time constraints in a real-time application like a game. Although neural networks can be useful in several areas in bot AI they are not used for the Quake III Arena bot.

3.6 Expert systems

An expert systems [7] is a system that stores human expertise, gained from training and experience. Such a system has three important aspects. The knowledge of facts, the knowledge of relations between the facts, and a heuristic or efficient method for storage and acquisition of this information. The construction of an expert system is called knowledge engineering. A knowledge engineer extracts the knowledge (procedures, strategies, information filters, common events, etc.) out of human experts and transforms this knowledge into an expert system. The key issue in expert systems is the way knowledge is stored in, and extracted from the knowledge base. Logic systems provide powerful tools to represent and infer knowledge in an expert system. Production rules or rule-based systems are therefore often used to implement an expert system. Production rules originate from the IF (…) THEN structure, which is known from traditional procedural languages. The rules consist of a condition side (the antecedent) and an action side (the consequent): IF (condition) THEN (action). The condition is a logical expression of facts from the knowledge base. The action either creates one or more new facts, removes facts or triggers certain events. The difference in rule-based programming as opposed to conventional programming lies in the fact that the statements in conventional programming languages are executed in a predefined order. Rule-based systems use an inference engine, which determines the rules to be fired based on the current facts. In this way new facts can change the course of the program as it proceeds to fire other rules, that are not necessarily stored consecutively to previously fired rules.
An expert system can be used to implement the reasoning of a bot. The expertise from human players is extracted and stored in a knowledge base. Production rules can be used to create new facts or initiate certain actions. For instance the bot could use the following production rule: IF the bot is fighting AND the bot is low on health AND the bot does not have a powerful weapon THEN retreat from the fight. The concepts “low on health” and “does not have a powerful weapon” are not clear facts in this example. Fuzzy logic as described in section 3.4 can be used to give a clear meaning to such concepts. A value of truth can be attached to the statement “low on health” based on the amount of health the bot has. In the same way a value of truth can be attached to the concept “the bot does not have a powerful weapon” based on the weapons and ammunition the bot is carrying. Using a fuzzy relation, it is also possible to combine the concepts “low on health” and “does not have a powerful weapon” into the new concept “the bot is not fit enough to fight”. The example production rule could then be replaced by: IF the bot is in a fight AND the bot is not fit enough to fight THEN retreat from the fight. A neural network as described in section 3.5 could also be used to give a clear meaning to the concepts “low on health” and “does not have a powerful weapon”.
An inference engine of some kind can be used in combination with production rules as in the examples. However, such production rules are also often just listed in several procedures. The Quake III Arena bot uses production rules to explicitly represent certain knowledge and make certain decisions. The usage of these production rules is described in section 15.

3.7 Genetic algorithms

A genetic algorithm (GA) is a search procedure that uses the mechanics of natural selection and natural genetics. A genetic algorithm was first developed by J.H. Holland in the 1960's. It uses evolutionary techniques, based on function optimization to develop better solutions to a problem. First a population of possible solutions to a specific problem is generated. The better solutions are then recombined with each other to form new solutions. These new solutions are used to replace the poorer of the original solutions. When creating new solutions they are often slightly mutated in order to keep the search domain somewhat larger, and perhaps find solutions that were not anticipated. This process of natural selection is repeated many times to acquire the best solution to the problem.
A genetic algorithm can be used on a collection of bots where each bot is slightly different. The process of recombining better bots to new ones, and replacing the bots that perform less good with the new ones, can be used to let a better bot evolve. Genetic algorithms can also be used to optimize subsystems used for the AI of a bot. For the Quake III Arena bot a genetic algorithm is used to optimize fuzzy logic for specific purposes. The experiments with this genetic algorithm are described in section 9.

4. Related work

4.1 FPS games & AI

Several first person shooter (FPS) games have been developed before Quake III Arena. The graphics engines of these games steadily advanced over the years. However such a continuous progression is not quite found for the artificial intelligence of the opponents in these games. Some of the most important and influential FPS games over the years are listed here.

Figure 4.1: Wolfenstein 3D, 1993 by id Software

In 1993 one of the first 3D shoot-em up games saw the light of day: Wolfenstein 3D. This game by id Software set a standard for FPS games to come. The player has to navigate through a maze-like map taking out enemies on the way. The enemies are usually on patrol through the maze and initiate an attack when the player is sighted. When the player runs away the enemies can chase the player but they are limited in their ways. They can open doors but usually try to reach the enemy navigating in a more or less straight line. As a result they easily loose track of the player in the maze.

In 1994, Doom set a new standard for FPS games. This game, also developed by id Software, has a much more advanced graphical engine than Wolfenstein 3D. The opponents however, are not notably smarter than the enemies in Wolfenstein. In this game the player also has to go through maps taking out numerous enemies. There are a variety of different opponents of which some are able to fly.

Figure 4.2: Doom, 1994 by id Software

Figure 4.3: Duke Nukem 3D, 1995 by 3D Realms

In 1995 3D Realms released the FPS game Duke Nukem 3D. The graphical engine in this game is slightly more advanced than the Doom engine. Not all the floors and ceilings are necessarily horizontal, and the player is able to swim in certain areas. The opponents however, are not notably smarter.

Quake was released by id Software in 1996. In this game the player has six degrees of freedom. In previous games the player was only able to turn left and right. The player can now also look up and down and the view tilts slightly when running around a corning. The opponents and items are modeled as true 3D objects instead of flat images (sprites) as used in previous games. The artificial intelligence of opponents is still relatively simple as in previous games. Some of the enemies patrol through the environment and initiate an attack when they sight the player or hear sounds from the player.

Figure 4.4: Quake, 1996 by id Software

Figure 4.5: Quake II, 1997 by id Software

Quake II was released late 1997 by id Software. This time around not only the graphics engine got more advanced. The opponents appear more intelligent than the opponents in previous games. The enemies are able to duck to avoid incoming missiles, and they also do a much better job at chasing when the player runs away from a battle.

In 1998 Unreal was released by Epic. This was the first game to ship with artificial players or bots. Such a bot can be used as a substitute for a human player in any of the multiplayer game modes. The bots appear considerably smarter than the enemies in previous games. To navigate through the game environments the bots use waypoints. These waypoints are placed throughout the environments. Using these waypoints the bots can find specific locations on the map and retrieve items and powerups to put up a better fight.

Figure 4.6: Unreal, 1998 by Epic

Figure 4.7: Half-Life, 1999 by Valve Software

In 1998 Valve Software released Half-Life. Although the graphics engine is not revolutionary the game does add to the genre of FPS games. In this game the player is up against both aliens and the army after a failed scientific experiment. The opponents seem able to team up with each other and team members can provide suppressing fire. Interesting are also the scientists and security guards in the game. They can follow the player around and often open secured doors if necessary. The security guards can also attack enemies when they come into sight. However in such an attack little attention is paid to friendly forces that might be in their line of sight. As a result security guards will often hurt co-workers or the player. The security guards do not seem to care much about their own lives as they never abandon a fight and keep shooting at enemies till either party dies. Overall Half-Life does bring some new and interesting artificially intelligent behaviour into the genre of FPS games.

In 1999 Unreal Tournament was released by Epic. This game is mostly a multiplayer game where in single player mode the player works his way through a tournament playing against bots, much like playing with human players on a network or online. These bots are quite sophisticated and do a pretty good job at simulating human players. The bots can also play the included team games like Capture the Flag (CTF), Assault and Domination. Just like the bots in Unreal these bots use waypoints to get around the levels.

Figure 4.8: Unreal Tournament, 1999 by Epic

4.2 Previous work

Before and during the development of the Quake III Arena bot there was little literature available on the subject of artificial players for FPS games. Due to the lack of a solid literature foundation a lot of experimenting was required for the development of the Quake III Arena bot. Unreal Tournament is a game very similar to Quake III Arena which also uses artificial players to provide a single player experience. However this game did not provide any reference material because it was released only shortly before Quake III Arena, and the development of both games was mostly concurrent. The previous development of the Omicron bot for Quake and the Gladiator bot for Quake II did however provide a solid base for tests and experiments. The experience gained from the development of these bots was used to create Quake III Arena bot.

4.2.1 Omicron bot

The Omicron bot [31] is an artificial player for the computer game Quake. Quake is, much like Quake III Arena, a first person shoot-em up game. However Quake is not solely focused on playing with other people on a network or the Internet. The game also includes a single player experience where the player goes through several dimensions eliminating numerous demons and other enemies.

Figure 4.9: An Omicron bot

The Omicron bot does not play the single player game. The bot was created to emulate a human player as an alternative to playing with other people on a network or online. The Omicron bot is very limited in both design and technology due to the language it was coded in. This language is QuakeC, which is similar to the commonly used programming language C. QuakeC is used to modify the game and implement the AI. QuakeC has no means to implement complex data structures and string manipulation is not possible. The code is interpreted by the game, which makes it relatively slow.

The bot has no prior knowledge about the environment when it enters the game and has to find its way around while playing. The bot wanders around and drops bread crumbs or waypoints as it goes. These waypoints allow the bot to find back locations along the route it followed.

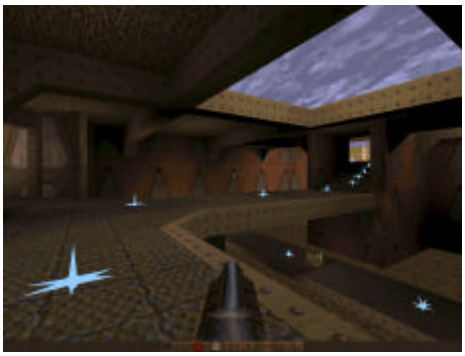


Figure 4.10: Quake map with waypoints represented by blue stars

After wandering through the whole map the bot should be able to travel to most locations on the map. However a lot of locations are still hard to reach for the bot at that point. The bot relies on very limited functionality to explore its environment. The bot is much like a blind man exploring his surroundings with a stick.

The Omicron bot comes with a variety of bot characters. Although these characters have not much more depth than their own outfit, name and personal chatter, human players often have the illusion that all bots have different personalities with different skills and abilities. The bot uses fuzzy logic to decide on how much it wants to have or do certain things. Fuzzy logic is also used for weapon preferences and to decide which items the bot wants most. These preferences are the same for all bot characters.

4.2.2 Gladiator bot

The Gladiator bot [32] is an artificial player for the computer game Quake II. This game is the successor of Quake. Quake II is much more focused on the single player experience with a compelling story line. However there is also a multi player experience.

The Gladiator bot is created, just like the Omicron bot, as an alternative to playing with other people on a network or the Internet. The bot does not play the single player game. Unlike the Omicron bot the Gladiator bot is not limited due to the language it is coded in, or due to how the bot code cooperates with the game program.

Figure 4.11: Quake II bots

A whole range of different Gladiator bot characters are available for people to play with. These characters have a lot more depth than just a different name and looks. The characters have their own play style and preferences. These preferences are stored in personal fuzzy logic for each bot character.

Several new systems have been developed for the Gladiator bot. The Quake III Arena bot uses and builds upon a lot of technology that has first been developed for the Gladiator bot.

5. Bot Architecture

5.1 Layered architecture

The Quake III Arena bot is build up in several layers. The awareness of decisions the bot makes while playing the game increases with higher layers. The decisions from higher layers are executed through lower layers. The layered structure of the bot is shown in figure 5.1.

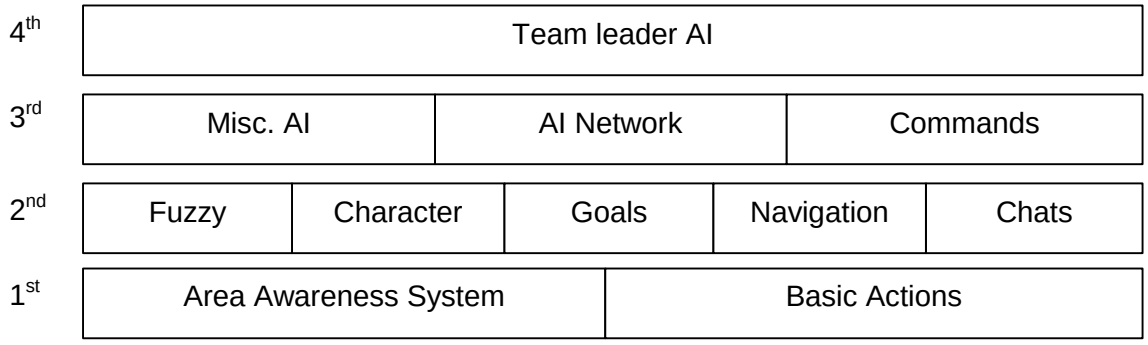


Figure 5.1: Layered architecture

The 1st layer is basically the input and output layer for the bot. The Area Awareness System is the system that provides the bot with all the information about the current state of the world. Information about the state of the world is received as a set of variables directly from the game program. For fast and easy access and usage, a lot of formatting and pre-processing is performed on this information. Everything the bot senses goes through the Area Awareness System.
The basic actions are the output of the bot. The output is formatted in a way that conforms to the parameters of the game. The output of the bot is the player input for the game, much like a human player uses a keyboard and a mouse.

The 2nd layer provides the intelligence that is often subconscious to skilled human players. This layer includes AI to select goals using fuzzy logic, AI to navigate towards a goal, AI to interpret chats, AI to construct chat messages and also functionality to store and retrieve characteristics of bots is included.

The 3rd layer is a mixture of production rules (if-then-else) and an AI network with special nodes for different situations and states of mind. This network is very similar to a state machine. All the higher-level thinking and reasoning takes place in this layer. This layer also contains a command module which allows the bot to understand orders and commands from other players or a team leader. The miscellaneous AI module contains AI to support behaviour used in, and decisions made in the AI network. This includes AI for the fighting behaviour of the bot and AI to navigate around obstacles and solve puzzles.

The 4th layer is the “brain” of the team leader or command center. In a team game one of the bots is designated to be the team leader and has this extra “brain” used to command

teammates. This allows the team leader to organize the team and accomplish tasks in a team.

The whole game typically runs in small time steps or frames, which is referred to as time-based simulation. Each frame the time and the whole game world advance a little bit. The ingame physics are also computed in a series of steps synchronous with the game. The bot's "brain" also operates in frames but not necessarily synchronous with the game. The bot's "brain" always operates at 10 Hz. Every tenth of a second the bot checks upon its status and situation and decides for the best actions to be taken. The bot uses the information available from the Area Awareness System to stay up to date about its status and the environment.

5.2 Information flow

Figure 5.2 shows the information flow between the layers.

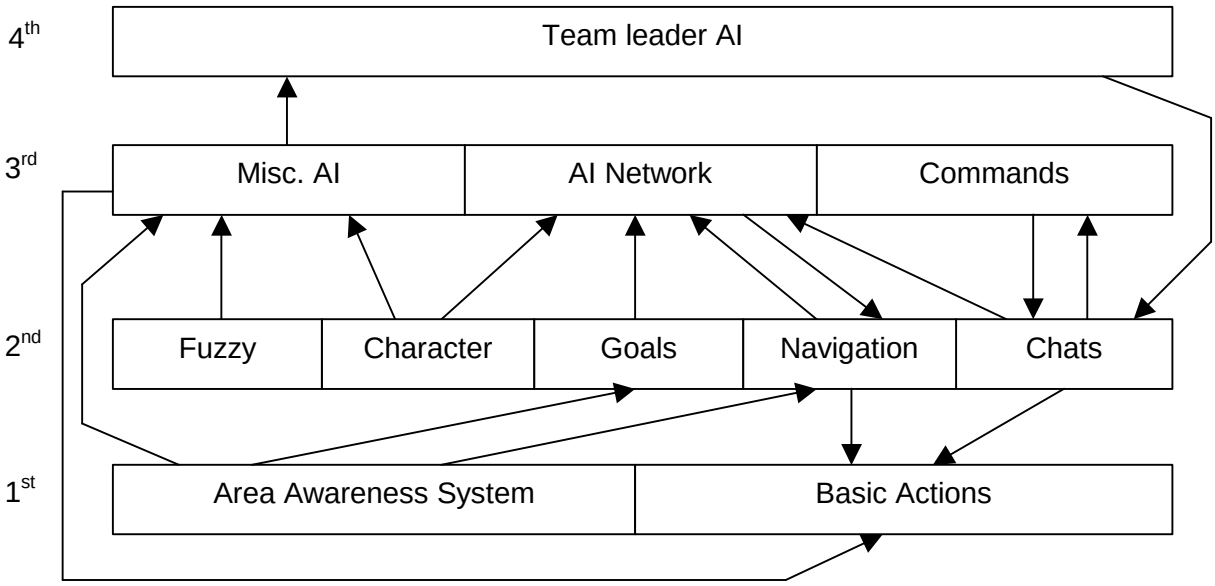


Figure 5.2: Information flow through layers.

All the arrows that go upwards in figure 5.2 represent the flow of information about the bot's status and it's environment. The bot uses this information to stay up to date on what is happening in the game world. The bot also uses this information to decide which actions should be taken in order to achieve certain goals in the game. This information usually becomes more abstract as it propagates towards higher layers. All the arrows that go downward represent the flow of information resulting from decisions the bot makes. These decisions and the resulting information eventually translate into a sequence of basic actions. These basic actions represent the player input of the bot.

Aside from the information flow between the four layers there is also limited communication between components within one layer. The AI Network in the third layer retrieves information from both the Miscellaneous AI and Commands component. In the second layer the Goals component retrieves information from the Fuzzy component for goal selection.

Code at higher layers requests information from lower layers. Information that is readily available can be used immediately. Calculations might be required to retrieve certain information. Such calculations are performed immediately and may not take up more than several milliseconds. If a calculation takes up too much time the game might hitch, because the game simulation only continues as soon as the calculation is completed.

5.3 Structure of game engine

The code of the game engine including the bot AI is structured as shown in figure 5.3.

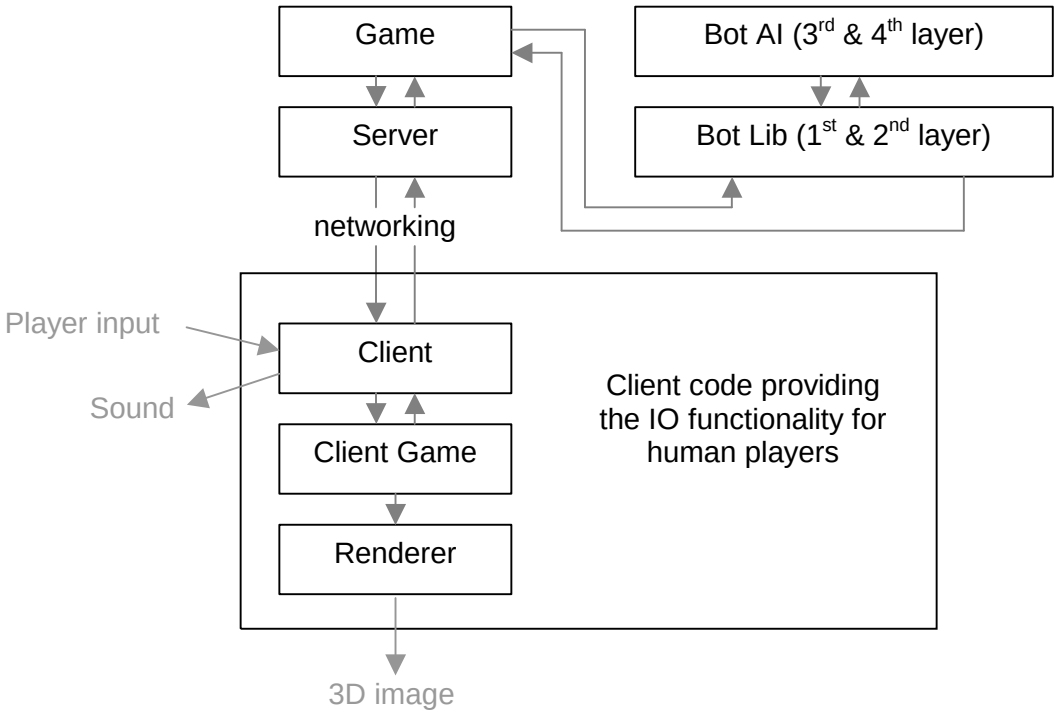


Figure 5.3: Integration of bot AI with the game engine.

The “Game” module sets the rules for the game and dictates how the game works. The “Server” module provides the functionality for players to connect to the game. The “Client”, “Client Game” and “Renderer” modules together provide the Input/Output (IO) functionality for a human player. The “Client” module records input from the input devices and sends it to the server. This module also forwards information from the server about the game, and what is visible to the player to the “Client Game” module. The “Client Game” module interprets this information and sends the necessary data to the “Renderer” which provides a 3D image to the player. The “Client Game” module also sends back information about sounds to the “Client”, which makes the sounds audible to the player.
In figure 5.3 the bot AI is shown in two parts: the lower two layers and the upper two layers. The “Game” module provides the Area Awareness System with all the necessary information about the state of the game world. This information consists mainly of entity data. The position, type, appearance etc. of entities in the game are communicated to the Area Awareness System. The bot input, a sequence of basic actions or intentions, generated by the bot, is sent directly to the “Game” module.

6. Area Awareness System

6.1 AAS

The Area Awareness System (AAS) is the whole system used to provide the bot with all the information about the current state of the world. This includes information about navigation, routing and also other entities in the game. All the information is formatted and preprocessed for fast and easy access and usage by the bot. The heart of AAS is a special 3D representation of the game world. All information provided to the bot is in some way related to or linked with this 3D representation.

A waypoint system is commonly used for routing and navigation purposes in 3D environments. The basics of a waypoint system are described first, in order to explain the basics of the 3D representation used for AAS. A waypoint system used for routing and navigation is a collection of nodes with links between them. For navigational purposes the links between these nodes have specific properties. The most important property is, that one can easily travel from one waypoint to another if they are linked. In other words the navigational complexity to reach one waypoint from another along a link is minimal, for instance the navigation along a straight line. A waypoint system with such a property is the result of a “divide and conquer” approach. Traveling from one waypoint to another is a sub problem with a simple solution. All places reachable from waypoints should now be reachable from any waypoint by traveling along one or more other waypoints. The disadvantage of using waypoints is that correctly determining if a point is reachable from a waypoint, or if a waypoint can be reached from a certain point, still involves complex and time consuming real-time calculations.

AAS has a similar property as the waypoint system. AAS uses 3D bounded hulls, called areas, with a specific property: the navigational complexity for traveling from any reachable point in an area to any other reachable point in the same area is minimal. In Quake III Arena this means a player can move between any such two points by just walking or swimming along a straight line.

Of course only knowing this property of each area does not provide all the information required for routing and navigation. However, so called reachabilities between areas can be calculated. Such a reachability is created from one area to another if a player can easily travel from one area to the other. Calculating these reachabilities is not all that difficult because a lot of areas will touch each other. When two areas touch, it can easily be verified if a player can really travel from one area to the other. This does not cover all the possible reachabilities between areas, but as will be shown later on, calculating other reachabilities is sometimes more complex, but definitely possible.
The system, as it is presented here, is primarily focused on navigation and routing. However a lot of other information can be retrieved from or linked to this 3D representation.

6.2 Creating areas

To satisfy the basic navigation property of AAS, areas will have to be created with minimal navigational complexity from any reachable point in an area to any other reachable point in the same area. A convex open space has this property. Convex areas do not have obstacles within them that can make the navigation more difficult. If a player can swim in an area, then the convex area always has minimal navigational complexity. However if the player has to walk through the area, then the requirement of the area being convex is not restrictive enough. The reason is that convex areas can still have gaps in the ground the player can fall into. However areas with gaps can be subdivided into multiple areas that do not have gaps. Basically all convex areas either have minimal navigational complexity or can easily be transformed into such areas. The areas do not have to be convex based on the visual geometry in Quake III Arena, but have to be convex for navigation. The basics of the collision detection in Quake III Arena will be outlined to acquire a better understanding of what convex for navigation actually means.

Collision detection

Quake III Arena uses bounding boxes for collision detection. The player resides inside such a bounding box and there is only collision with this bounding box. The detailed player model that is visible in the game is not used for collision detection. The bounding boxes are axial; the bounding planes of the box are aligned with the coordinate axes of the game world. The bounding boxes are never rotated so the bounding planes always stay aligned with the coordinate axes. The blue outlines in figure 6.1 show a bounding box. The red cross inside the bounding box is the origin of the box. In the picture the bounding box of a player is shown. However a bounding box of any size can be used for collision detection in Quake III Arena.
The Quake III Arena world is built up using so called brushes, which are convex building blocks. A brush is a volume bounded by a number of planes. The space contained between these bounding planes is convex. The normal vectors of the bounding planes of a brush point outward. For instance, the green cube in figure 6.1 is a brush with 6 bounding planes. In Quake III Arena there is only collision between bounding boxes and these convex building blocks.
Collision detection calculations can be simplified because convex brushes are used, and the bounding boxes are always axial. Instead of calculating the collision between a bounding box and a brush, the collision between an expanded brush and the origin of the bounding box can be calculated. For the expansion of a brush the bounding planes have to be moved appropriate distances along their normal vectors. The expansion distance for a brush bounding plane depends on the orientation of the plane.

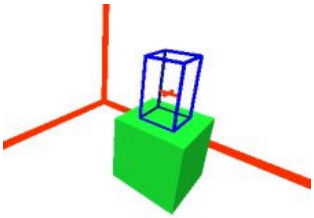


Figure 6.1: Bounding box on cube shaped brush.

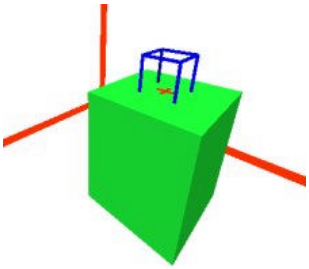


Figure 6.2: Expanded cube shaped brush.

In figure 6.1 a bounding box is standing on top of a cube shaped brush. In figure 6.2 the cube shaped brush has been expanded and now the origin of the bounding box is on top of the brush.
The expansion distance along the normal vector of a brush plane, is the smallest distance between the bounding box origin and the brush plane, when the bounding box touches the brush plane. This distance is shown by the dashed line in figure 6.4 and can be calculated as follows. Assume there are two vectors. A vector 'mins', which has the minimums of the bounding box relative to the bounding box origin (blue arrow in figure 6.3). And a vector 'maxs', which has the maximums of the bounding box relative to the bounding box origin (red arrow in figure 6.3). Depending on the orientation of the brush plane, one of the corners, edges, or sides of the bounding box will collide with the plane first.

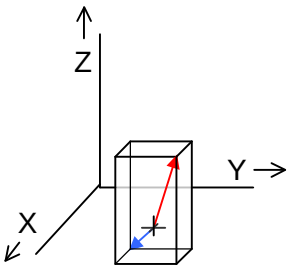


Figure 6.3: 'mins' and 'maxs' vector in bounding box.

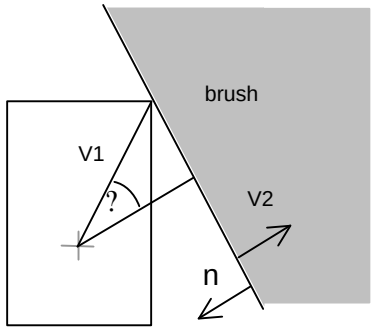


Figure 6.4: 2D view of bounding box colliding with brush.

A new vector is constructed between the bounding box origin and a point on the brush plane where the bounding box first touches this plane. This vector, called 'v1', can easily be derived from the 'mins' and 'maxs' vector and the orientation of the brush plane. The normal vector of the brush plane is denoted with 'n' in figure 6.4.

Now let 'v2' be the inverse of the brush plane normal vector. And let '?' be the angle between the vectors 'v1' and 'v2' then the following holds:

(v1 * v2) / (|v1| * |v2|) = cos(?) = expansion distance / |v1|
(v1 * v2) / |v2| = expansion distance
v2 is normalized and therefore |v2| = 1
v1 * v2 = expansion distance

A problem is encountered when all brushes are simply expanded and the bounding box origin is used to collide with these expanded brushes.

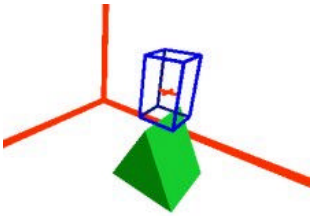


Figure 6.5: Bounding box on wedge.

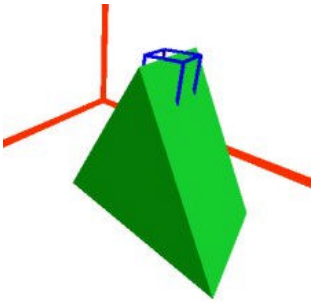


Figure 6.6: Expanded wedge.

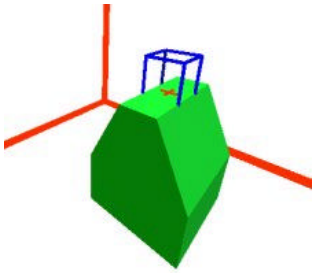


Figure 6.7: Beveled wedge.

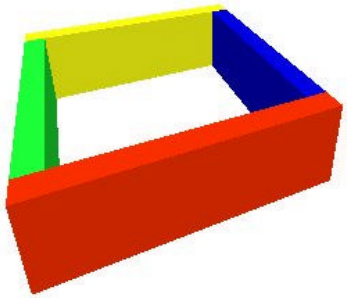
In figure 6.5 a bounding box is standing on top of a wedge shaped brush. When the bounding panes of this wedge are expanded the wedge as seen in figure 6.6 is obtained. Clearly can be seen, that the wedge expanded too far in certain places. The origin of the bounding box is contained within the wedge, where it should rest on top of the wedge after expansion. Why does the expansion not work with the wedge while it worked fine for the cube? The cube had only axial planes where the wedge does not. When two non-axial planes are expanded, the edge that they create is expanded more than it should. Expansion like that can be prevented by adding additional bounding planes to the wedge shaped brush. These additional bounding planes will not change the shape of the brush, because they are only added at the edges and/or corners of the brush. These additional bounding planes will be called brush bevels. The required bevels are all axial planes going through an edge or corner of the brush that are not already part of the brush. Also along edges all planes that are parallel to one of the coordinate axes and do not change the shape of the brush need to be added. When all these bevels are added to the wedge shaped brush and the brush is expanded including all the brush bevel planes, the brush as shown in figure 6.7 is obtained. As can be verified, this expanded brush has the shape, which should be used for collision with the bounding box origin.

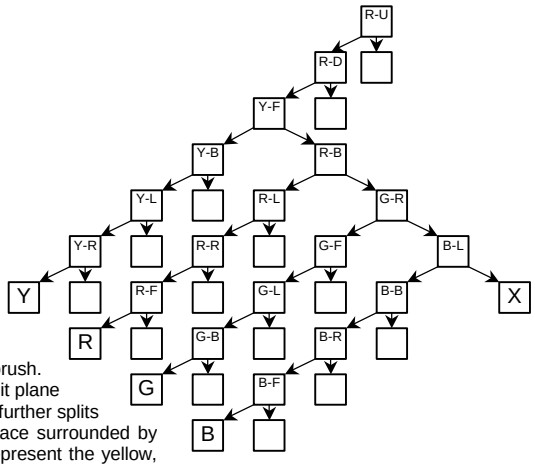
Besides brushes, Quake III Arena has another primitive used to build maps. This primitive is the Bezier curve. [12] With these curves nicely rounded shapes can be visualized in the maps. In Quake III Arena the curves are tesselated and the polygons that approximate the curve are transformed into (invisible) brushes for collision detection. Actually any polygon can be transformed into a brush. After adding bevels to these brushes, they can also be expanded and easily used for collision detection. A brush created from a single polygon has no volume, but is perfectly valid for expansion. The first bounding plane of this brush is the plane going through the polygon boundary points and the second boundary plane is the opposite of the first. The brush also has a bounding plane for each edge of the polygon. Any additional bevel planes are added as well.

Creating areas

All primitives used for collision in Quake III Arena can now be expanded. All the space outside these expanded solid brushes is the space, where players can move around i.e. where the origin of the player bounding box can be. At this point convex hulls can be defined within this space. Within these convex hulls the player will be able to move with minimal navigational complexity. These convex hulls will become the areas of AAS.

Binary Space Partitioning is used to create these convex hulls or areas of AAS. H. Fuchs, Z. Kedem, and B. Naylor introduced Binary Space Partitioning (BSP) for graphics rendering purposes [12]. With binary space partitioning a tree structure is created, a so-called BSP tree. This BSP tree is a binary tree, which represents the entire space, an entire map in the game. Each node in the tree represents a convex subspace and stores a plane, which splits the space the node represents in two halves. A node also stores references to two other nodes, which represent each half. These two nodes are often called children or child nodes.





The bounding planes of the brushes as shown in the image have names that start with the first letter of their color followed by the first letter of the side: front, back, left, right, up or down respectively. For instance Y-F stands for the front of the yellow brush. All nodes that split space in two halves have the name of the split plane used. All nodes without name are convex sub-spaces where no further splits could be made. The node with the name X represents the space surrounded by the four brushes. The nodes with the names Y, R, G and B represent the yellow, red, green and blue brush respectively.

Figure 6.8: BSP tree of four brushes.

The BSP tree created for AAS uses the bounding planes of the expanded brushes as split planes at the nodes. At each node one of these bounding planes is used, which splits the space represented by the node in two halves. Only a bounding plane of an expanded brush side, that is totally or partly in the space represented by a node, may be used as splitter for that node. A node may not use planes as splitters, which are found, when going from the node back to the root of the tree. No further splits are made at a node when there are no sides from expanded brushes inside the space represented by the node. When no further splits can be made at a node, the planes that are found when going from this node back to the root of the tree, define the convex hull the node represents. Nodes where no further splits can be made, will be called leaf nodes. A selection of these leaf nodes will become the areas of AAS.

Many such BSP trees can be created from the same set of expanded brushes. The tree best suitable for AAS is the one that defines the smallest number of convex hulls. While building the tree, for each node the split plane will be chosen that will minimize the number of convex hulls created. A BSP tree with a minimal number of convex hulls is best suitable for AAS, because routing between areas has to be calculated. This will be explained in detail in section 6.5.

Constructive Solid Geometry

The more bounding planes from expanded brushes there are, the more potential splits are made in the BSP tree. More splits usually also means more convex hulls. A lot of potential splitters can be removed before the BSP tree is built.

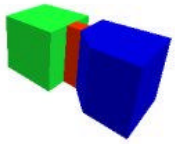


Figure 6.9: Three brushes

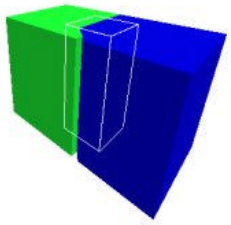


Figure 6.10: Three expanded brushes with overlap.

After expanding all the primitives used for the collision detection in Quake III Arena there are a lot of expanded brushes that either overlap or are contained within other brushes. Figure 6.9 shows three brushes before expansion. Figure 6.10 shows how after expansion one of the brushes is fully contained within the other two brushes, and also shows how the brushes overlap. All the contained brushes and brush overlap can be removed. Contained brushes can be removed because a bounding box will never collide which such brushes. Also one of the overlapping parts of brushes that overlap can be removed, because a bounding box only needs to collide with one of the parts. The removal of contained brushes and overlapping parts can be accomplished with simple Constructive Solid Geometry (CSG) [12] operations, because the expanded brushes are convex. After removing contained brushes and overlapping parts there are less brush bounding planes that need to be used as splitters in the BSP tree.

More bounding boxes

So far the assumption has been made that the player is only using one bounding box. This is not the case. When a player crouches, another, smaller bounding box is used for collision detection calculations. This does not really provide a problem for AAS though. Multiple sized bounding boxes can be compiled into the same BSP tree. For each bounding box a set of brushes is created and expanded for that bounding box. CSG operations are only performed on and between brushes that are expanded for the same bounding box. From all sets of expanded brushes one BSP tree is created. After creating the BSP tree it can easily be determined which bounding box(es) can move in each convex sub-space represented by a leaf node. When a convex sub-space represented by a leaf node contains no expanded solid brushes then all bounding boxes can move around there. When a convex sub-space represented by a leaf node contains one or more solid brushes expanded for a certain bounding box, then that bounding box cannot move around there.

Contents of volumes

In Quake III Arena the contents of certain volumes is also defined with brushes. For instance a water volume is defined with a special brush. The same goes for lava and slime. These brushes that define a contents can also be expanded and compiled into the BSP tree. The leaf nodes that only contain (parts of) these brushes, that define content, can be marked as volumes with that content.

Solid sub-trees

The BSP tree now defines convex hulls (areas). Before another representation for these convex hulls is created, the BSP tree can be optimized. The BSP tree contains completely solid sub-trees. When the sub-space represented by a node contains only solid expanded brushes and there is no “open” space between them, then the whole node is considered solid. For AAS only the areas where the player can move around are interesting. Obviously the player cannot move around in a completely solid sub-space. The node that represents such a solid sub-space can still have child nodes. These child nodes represent solid sub-spaces within the solid space represented by the node. These solid sub-spaces are not interesting for AAS, and the tree structures that represent them can be removed. Removing all solid sub-trees can make the BSP tree significantly smaller. In some cases, it can more than half the total number of nodes in the BSP tree.
There are several causes for the existence of solid sub-trees within the BSP tree. One of them is shown in figure 6.11.

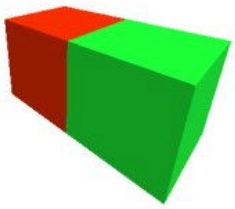


Figure 6.11: Two adjacent brushes.

A BSP tree is created from these two expanded brushes and the plane used at the last node of the tree is the plane that separates the two brushes. This node represents a fully solid sub-space and it’s children represent the original brushes. Since the node is already fully solid there is no need to store its children. Especially after the CSG operations are performed on the expanded brushes this situation is quite common.

Portalisation

In order to calculate reachabilities and routes between areas (as will be done later on), another representation is required for the convex hulls (areas). The BSP tree does store all the information needed, but the representation cannot easily be used to calculate relations between areas. A representation with areas that are bounded by faces would be much more suitable. These faces are polygons that either represent solid walls or lead to other areas. With such a representation adjacency of areas can easily be determined, and it is easier to find or calculate geometric properties. Such a representation can be created by portalising the BSP tree. This technique creates

portals between all the leaf nodes defined by the BSP tree. These portals are the faces that bound the areas.
After portalisation the basic representation needed for AAS is created. However this portalised BSP tree still needs some work and the whole representation can be optimized in several ways. The BSP tree structure is also not thrown away at this point, because it is a very useful access structure to the areas of AAS, as will be shown later on. Each area, with the face boundary representation, is linked into the BSP tree at the node that represents its convex sub-space.

Outside maps

A map in Quake III Arena is a space enclosed by brushes. There is no way for a player to go outside this enclosed space. Now that adjacent areas can easily be found through the created portals, it is possible to flood through these portals. When starting from all known valid positions inside the map, the convex hulls these positions are in can be found. Flooding through the portals of these convex hulls all other convex hulls or areas that are also within the enclosed map space can be found. If all these convex hulls are marked then all convex hulls that are not marked can be eliminated because those convex hulls are outside the map, and a player can never move around there.

Gravitational subdivision

As mentioned earlier not all convex areas satisfy the basic navigation property required for AAS. All areas that contain water, or another liquid the player can swim in, satisfy the basic navigation property. However areas a player walks in can have gaps in the floor. Basically if a player is able to stand somewhere in an area, there should not be any gaps in the floor in the same area. If an area does have gaps the player can fall in the gaps, and the player might not be able to move in a straight line between any two reachable points in the area. When an area, with both places where the player can stand, and one or more gaps in the floor is found, this area needs to be split into several areas. Every time such an area is found it is split with a vertical plane that goes through one of the edges of the gap. These split planes will be called gravity planes. Instead of this area a new node is added to the BSP tree that stores the new split plane. The new node will have the two new areas, created after splitting the original area, as it's children. All areas, that have both places where the player can stand and gaps in the floor, are split with gravity planes until no such areas are left.

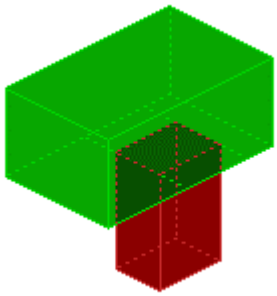


Figure 6.12: Area with gap.

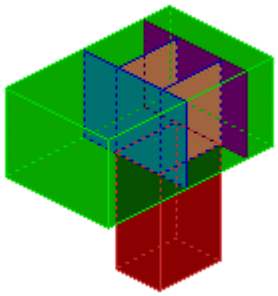


Figure 6.13: Area subdivided around gap.

In figure 6.12 the green area has a gap in the floor. The gap leads to the red area. In figure 6.13 the green area is subdivided into multiple areas, using gravity planes through the edges of the gap.

Merging areas

The BSP tree is already optimized to define the least number of convex hulls or areas. In addition to that, there is another way to reduce the total number of areas. As long as the new area satisfies the navigation property of AAS, two adjacent areas can be merged into one new area. Each pair of adjacent areas is tested and merged if and only if the new area is still convex, and no gaps are introduced to an area in which the player can stand. When two areas are merged the BSP tree is modified accordingly. This can cause multiple branches of the BSP tree to point to the same area.

Melting things together

The data used for AAS is the collection of all the areas with their face boundary representation and the BSP-tree as a fast and very useful access structure to the areas. All the boundary representations of the areas will share data. The areas will share faces, edges of faces and vertices. This will allow to more easily find shared faces, edges and vertices between adjacent areas.

6.3 Environment sampling

There are various ways to extract information from the 3D representation used for AAS. Especially the BSP tree is a very useful structure, which allows to calculate and extract certain information about the environment very easily.

Finding the area a player is in

First of all it will be useful to know which area a bot is in. Using the BSP tree there is a very fast and easy way to calculate the area a bot is in. One can start at the root node of the BSP tree and calculate the side of the plane, stored at that node, the origin of the bot's bounding box is at. Depending on the side of the plane the origin is at, one continues with one of the child nodes that represents either the sub-space at the front, or the sub-space at the back of the plane. At this child node one again calculates which side of the plane, stored at the child node, the origin of the bounding box is at, and one continues with one of it's children accordingly. This procedure is continued until one of the areas of AAS is found. That area is the area the bot is situated in.
The planes at the nodes are stored as a normal vector with a distance. This makes it rather easy to determine at which side of the plane a point is located. The side of the plane a point is at, can be determined from the sign of the dot product of the plane normal vector and the point, minus the plane distance. If the value is positive then the point is at the front of the plane. If the value is negative, the point is at the back of the plane, and if the value is zero then the point is on the plane. If the origin of the bounding box is on one of the split planes at a node, then one also continues with the child node that represents the sub-space at the front of the split plane.

Recursive subdivision by the BSP tree

Collisions of the bot's bounding box with the world can also easily be calculated using the AAS data structures. The collision with the environment of a bot moving along a straight line from a start to an end position can be calculated using the BSP tree.

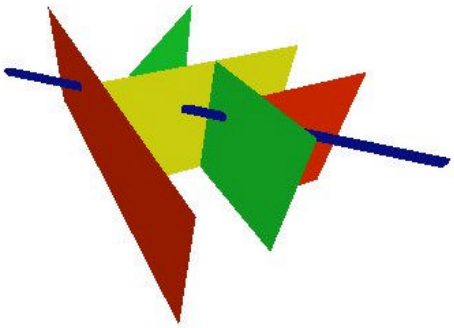


Figure 6.14: Trace subdivided by a BSP tree.

The collision can be calculated with recursive subdivision by the BSP tree of the line along which the bot moves. The movement along a straight line from a start to an end position will be called a trace. Figure 6.14 shows how a trace is subdivided into several trace segments by the planes of a BSP tree.
One starts at the root node of the BSP tree and calculates which side of the plane, stored at that node, the trace is at. Depending on the side of the plane the trace is at, one continues with one of the child nodes that represents either the sub-space at the front or the sub-space at the back of the plane. In case the plane stored at the node splits the trace, one continues with both children. At each child node one continues with only that segment of the trace that is inside the sub-space represented by the child node. At the child nodes one again calculates which side of the plane, stored at the child node, the trace or trace segment is at, and one continues with one or both of its children accordingly. This procedure is continued until all the areas of AAS and solid leaf nodes the trace goes through have been found. When the trace is split into two trace segments in this recursive process, one always continues first with the child node that contains the trace segment closest to the start of the trace. This way one ends up with a sorted list of trace segments from the start to the end of the trace. As soon as a line segment in this sorted list enters a solid sub-space there is a collision at the start of that line segment.

Finding the areas a trace goes through

Just like collisions of a bot's bounding box with the environment are calculated, it can also be calculated through which areas the movement along a straight line goes. Recursive subdivision by the BSP tree of the trace is also used here. The trace is chopped up by the BSP tree into one or more trace segments that go through certain areas. These areas can easily be listed and used for numerous things.

Areas a bounding box is in

In order to calculate if the bot's bounding box can or will touch the bounding boxes of other entities in the world, it is often useful to know in which area(s) the bounding boxes of entities are. To calculate this the bounding box of the entity has to be expanded, just like the brushes are expanded for collision calculations. This expansion is necessary, because the not expanded bounding box can be outside all areas, when at the same time the bot might be able to touch the bounding box while standing in a certain area. After expansion it can be calculated in which area(s) the bounding box resides, by testing on which side of the BSP tree split planes the bounding box is situated. This works similar to how the area a bot is in, is found. However a bounding box is now used instead of a point. One starts at the root node of the BSP tree and calculates which side of the plane, stored at that node, the bounding box is at. Depending on the side of the plane the bounding box is at one continues with one of the child nodes, that represents either the sub-space at the front or the sub-space at the back of the plane. In case the plane stored at the node splits the bounding box one continues with both children. At each child node one again calculates which side of the plane, stored at the child node, the bounding box is at, and one continues with one or both of it's children accordingly. This procedure is continued until all the areas of AAS the bounding box is in are found.

6.4 Reachability

Just the area representations are not sufficient for the bot to travel through the whole map. The bot will need to know how to travel from one area to the other, if possible at all. Therefore so-called reachabilities between areas are calculated. Such a reachability always starts in a certain area and leads to one other area. All possible reachabilities can be classified using about 12 different types. The different types used in Quake III Arena are listed below:

- Swimming in a straight line
- Walking in a straight line
- Crouching in a straight line
- Jumping onto a barrier
- Walking of a ledge
- Jumping out of the water
- Jumping
- Teleporting
- Using an elevator
- Using a jump pad
- Using a bobbing platform
- Rocket jumping

The reachabilities are calculated for each area, and they link the area to another area. They also store a start and end point of the movement along the reachability. The reachabilities often also store a reference to the area boundary face and/or edge that leads to the other area. Below the area a reachability is created for will be called, A1. The area a reachability leads to, will be called A2.

Swimming in a straight line

The ‘swim’ reachability is one of the easiest to find in the AAS data. From each area filled with water, or another liquid the player can swim in, one can look at the faces that lead to other areas also filled with a liquid the player can swim in. A ‘swim’ reachability is then created from the area to the other area which is found through the face.

Walking and crouching in a straight line

Most of the ‘walk’ reachabilities are also very easy to find. From each area one can look at adjacent areas through the faces. When the player can walk in both areas, and there are floor planes in both areas that meet at an edge, then a ‘walk’ reachability can be created. Finding ‘crouch’ reachabilities is very similar to how the ‘walk’ reachabilities are found. When the player has to crouch in either one of the two adjacent areas a ‘crouch’ reachability is created.

Several reachabilities

Boundary faces of an area on which the player can stand will be called ground faces. Several reachabilities can be calculated using the edges of these ground faces. A vertical plane is created through a ground face edge that is not shared with other ground faces in the same area. Now one searches for edges from ground faces of other areas that are on this same plane. When an edge is found to be on the same plane, the shortest distance between that edge, and the edge used to create the plane is calculated. Several different reachabilities can be created depending on this distance. The edge used to create the plane will be called E1, and the edge found to be on the same plane, E2. The figures 6.15 through 6.17 are side views, and the edges E1 and E2 are coming towards the viewer. The vertical line represents the shortest distance between the edges E1 and E2. In all three cases the edge E2 is positioned higher relative to the edge E1. Also in all three cases the shortest distance between the two edges is smaller than the maximum step height. Players can walk up stairs automatically if the step height is smaller than a certain number of units. Figure 6.15 shows a normal step. Figure 6.16 shows a step with shallow water in area A1. Figure 6.17 shows a step with water up to the step.

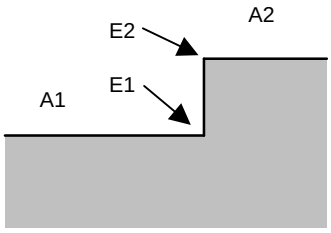


Figure 6.15: Step.

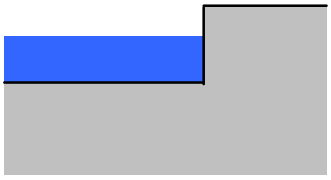


Figure 6.16: Step with low water.

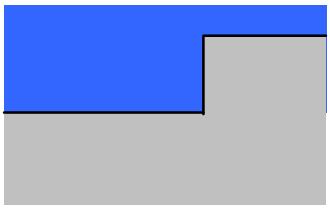


Figure 6.17: Low water onto step.

In all three cases the water is too shallow to swim in and a ‘walk’ reachability will be created.

In case edge E2 is positioned higher than the maximum step height relative to the edge E2, one checks if a player can jump up onto the barrier. The maximum height of a barrier, a player can jump onto, is a fixed value in the game physics. If the edge E2 is not higher than this maximum height relative to E2, a ‘barrier jump’ reachability is created.

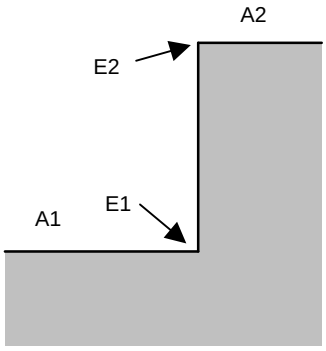


Figure 6.18: Barrier.

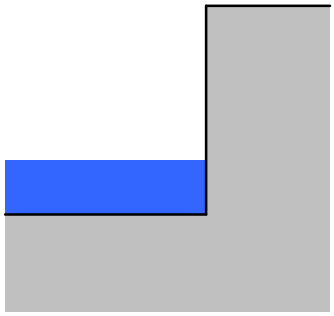


Figure 6.19: Barrier with low water.

Figure 6.18 and 6.19 show two cases where a ‘barrier jump’ reachability would be created. The reachability is also created when area A1 is filled with water that is too shallow to swim in as shown in figure 6.19.

The edge E2 can also be positioned lower relative to the edge E1. If the edge E2 is not lower than the maximum step height relative to edge E1 a ‘walk’ reachability is created. The player will just walk down the step and/or stairs. In case the edge E2 is lower than the maximum step height relative to the edge E1 a ‘walk off ledge’ reachability is created. The player will not be able to walk back up, the player will have to jump back up or may not be able to get back up directly at all.

A ‘walk off ledge’ reachability reachability is also created when the area the reachability will lead to is filled with water a player can swim in.

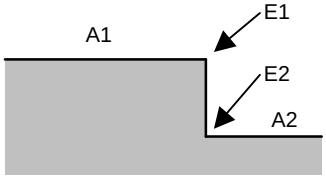


Figure 6.20: Step down.

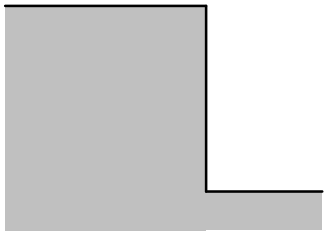


Figure 6.21: Ledge.

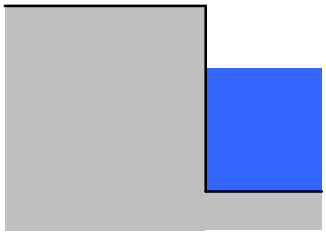


Figure 6.22: Ledge into water.

Figure 6.20 shows the situation where the player walks down a step and/or stairs. Figure 6.21 and 6.22 show cases where a ’walk off ledge’ reachability is created.

When the edge E2 is lower than the maximum barrier height, relative to the edge E1, one will have to make sure there are no solid objects the player might bump into when walking off the ledge. If the edges are far enough apart, obstacles as shown in figure 6.23 can obstruct the player.

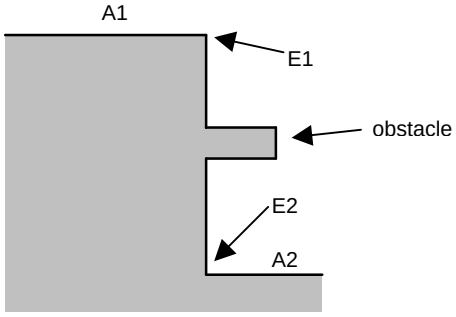


Figure 6.23: Ledge with obstacle.

A player bounding box is traced from edge E1 to edge E2 to find such obstacles. In case obstacles are found no reachability is created. When looking back at the reachabilities created for steps and barriers one will notice that no such check for obstacles is used there. The reason is, that the step and barrier height are smaller than the vertical size of the player bounding box. Due to the expansion of the solid brushes in the map before compiling them into the BSP tree, there simply cannot be any obstacles along that height.

Jumping out of the water

If none of the above reachabilities were created from area A1 towards area A2 one tests for a ‘water jump’ reachability. Players are able to jump out of the water onto a floor at the side of the water if the floor is not too high. Such a ‘water jump’ reachability is only created if area A1 is filled with water, and area A2 is either not filled with water or the water is too shallow to swim in. Figure 6.24 and 6.25 show these situations.

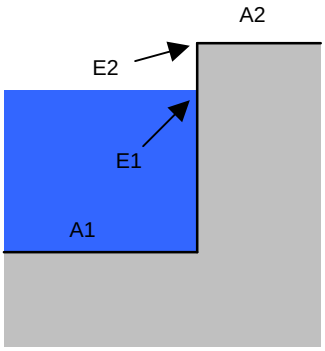


Figure 6.24: Water jump.

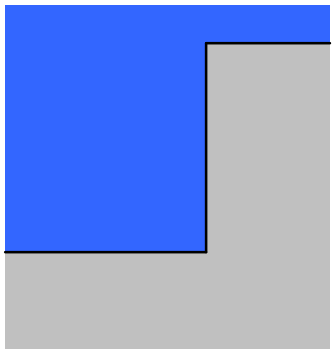


Figure 6.25: Water jump with low water onto floor.

To test if the floor at the side of the water is not too high, the distance between the water surface and the floor will have to be calculated. The water surface is also represented by one or more area boundary faces because water brushes are also compiled into the BSP tree (section 6.2). In figure 6.24 an edge of this water boundary face is denoted with E1. The edge of a ground face of the area the ‘water jump’ reachability will lead to,

is called E2. A ‘water jump’ reachability is created when the shortest distance between these two edges is smaller than the maximum height a player can jump up to out of the water.

Jumping

To find ‘jump’ reachabilities one looks once again at the edges of ground faces of the areas of AAS. The two closest points on the ground of two areas is searched for. One of the points will be on an edge of a ground face of area A1 and the other on an edge of a ground face of area A2. In case there is a range of closest points, the point in the middle of this range is used. Between these two points there must be one or more gaps. Such gaps are found by tracing a player bounding box down at the points on the edges.

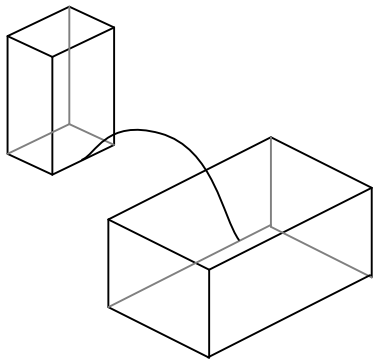


Figure 6.26: Jump reachability.

If the gaps exist a potential jump is predicted from area A1 towards area A2. Basically the full player movement along the jump arch is predicted to see if the player would bump into any obstacles. When no obstacles are found the ‘jump’ reachability is created from area A1 to area A2. If the two points on the edges of ground faces of area A1 and A2 are very close to each other, or the point on the edge of area A2 is situated a little bit lower, then players sometimes do not even have to jump. The player can bridge the gap(s) between area A1 and area A2 by running. In such a case a ‘walk off ledge’ reachability is created instead of a ‘jump’ reachability.

Teleporting

There are teleporter entities in certain maps. These entities come with a trigger brush. When this trigger brush is touched the player is teleported instantly to another location on the map. These teleporter brushes are compiled into the BSP tree just like water brushes. The leaf nodes or areas that contain only these or part of these brushes are then marked as areas that will teleport the player when entered. For each teleporter entity one searches for the areas that were created by the teleporter brush. Such areas are found by retrieving all the areas that are within the smallest bounding box that contains the teleporter brush, and only selecting those areas that are marked as containing the teleporter. From all these areas, ‘teleporter’ reachabilities are created towards the area that the destination of the teleporter entity is in. Portals that teleport a player work exactly the same.

Using an elevator

Quake III Arena also has elevator entities in certain maps. The bottom and top position of the elevator are stored with these entities. At these two positions one looks around for nearby areas the player can walk towards when standing on the elevator, or walk from onto the elevator. At both positions such areas can be found by tracing in several directions using the environment sampling functionality described in section 6.3. An elevator goes up automatically when a player is standing on top of it. ‘elevator’ reachabilities are only created from areas nearby the bottom position towards areas nearby the top position.

Using a jump pad

There are also jump pad entities in a lot of maps. These entities also come with a trigger brush just like the teleporter entities. When a player touches such a trigger brush the player is pushed in a specific direction with a certain velocity, much like an oriented trampoline. The jump pad brush is compiled into the BSP tree just like the teleporter brush and water brushes. The leaf nodes or areas that contain only these or part of these brushes are then marked as areas that will push the player into a specific direction. For each jump pad entity one searches for the areas that were created by the jump pad brush. Such areas are found by retrieving all the areas that are within the smallest bounding box that contains the jump pad brush, and only selecting those areas that are marked as containing the jump pad. From all these areas ‘jump pad’ reachabilities are created towards the area that the player ends up in, after being pushed by the jump pad and having flown through the air. The area towards which the reachability is created is found by predicting the player movement from the jump pad to where the player lands on the ground. The area the player lands in will be the area towards which the reachability is created. Acceleration pads work exactly the same as jump pads.

Using a bobbing platform

The bobbing platform is an entity that continuously moves up and down or horizontally back and forth. Players can often stand on top of these, to travel to other locations on the map. The positions between which the platform moves up and down or back and forth can be calculated from the entity. At these positions one looks around for nearby areas from which the player can move onto the platform or towards which the player can move from the platform. Such areas are found by tracing a bounding box at the edges of the platform. Between all combinations of areas at both positions, reachabilities are created both going up and down or back and forth.

Rocket jumping

In Quake III Arena a player can perform a so called rocket jump. The player shoots a rocket at the floor, and jumps at the same time. The player of course gets damaged by the rocket but does not necessarily get killed. The rocket explosion also pushes the player upwards. In combination with the jump the player can jump up to very high places. For the bots, ‘rocket jump’ reachabilities are only created towards areas that have one or more items in them. Many ‘rocket jump’ reachabilities can be created and a lot of them are of no interest to the bot. Also it would not be good to make the routing (see section 6.5) slow due to a lot of ‘rocket jump’ reachabilities, which the bot will not use most of the time. ‘rocket jump’ reachabilities are also only created from the center of

areas the player can stand in. This is simply to choose one of many possible start points for the rocket jump. From this center the upward velocity is calculated and the player movement is predicted towards areas that have items in them within range. When, in the prediction, the player is not obstructed by obstacles, and the player ends up in the right area, the 'rocket jump' reachability is created.

Complexity of calculations

Calculating all these reachabilities seems to involve rather time consuming operations. Testing each area against every other area for reachabilities is of $O(n^2)$ on the number of areas. However in a lot of cases there is no need to test each area against every other area. The ability to flood through the faces of one area towards other areas can speed up calculations considerably. For some reachabilities like the 'jump' reachability, this method cannot be used though. The 'jump' reachability is probably one of the most complex and time-consuming reachabilities to find. However since players can jump only so far and only jump up to a certain height a lot of areas can be discarded before trying to create 'jump' reachabilities towards them. Knowing the bounds of each area can speed up the search even more. The same of course goes for 'rocket jump' reachabilities.

6.5 Routing

Real-time vs. fixed pre-calculated tables

Having both the area descriptions and reachabilities, routes between areas can be calculated. A pre-calculated table with routing data is very fast for lookup in real-time, but the table will also take up quite some memory. The Quake III Arena environment hardly changes, so a pre-calculated table will work in most cases. However there are also quite a few situations where routes change due to small but significant changes in the environment. For instance doors that only open by pushing a button located elsewhere in the map can change the route towards a goal. To be able to deal with these situations a real-time dynamic routing algorithm is used. Calculating routes on the fly consumes quite some more calculation time than a simple lookup table. However in return a lot of flexibility is obtained. Because the Quake III Arena environment does not change continuously, part of the real-time calculated routing data is temporarily stored or cached for fast look up. With a restricted amount of the routing cache the memory requirements can be minimized.

Conventional algorithms

There are several algorithms readily available to calculate routes and travel distances and/or times in a graph. Dijkstra's algorithm and A* are two examples [11]. However, using these algorithms a problem is encountered. The Quake III Arena maps are usually quite complex and detailed. As a result the number of areas that are created for a typical map, is rather high. Maps with 5000 areas or more are not uncommon at all. Not all areas will be used for routing purposes. Only areas a player can stand or swim in are used. Still there will be a lot of areas used for routing. Algorithms like Dijkstra's or A* are simply to slow for usage in real-time with such large area counts. Especially when there is a game engine running at the same time, which consumes quite some calculation time. Another approach is required than using one of these conventional routing algorithms.

Multi-level algorithm that calculates cache

A multi-level routing algorithm is used. This routing algorithm calculates routing data with the same accuracy as conventional routing algorithms. There are no significant drawbacks compared to these commonly used routing algorithms. However the multi-level routing algorithm often requires significantly less calculation time and storage space. The routing algorithm always calculates and caches routing data for a specific goal area. The routing cache stores per goal area, the travel times of areas towards this goal, and the first reachability to be used from these areas towards this goal. The algorithm does not store the travel times towards goals per source area. The routing cache is stored per goal area, and not per source area because there are a lot of goals in Quake III Arena that stay at fixed locations. At the same time the bot will be moving around a lot and thus the source area will change a lot. Of course there are also goals that move around, for instance another player when being chased by the bot. However when the bot chases another player both the goal and source area change continuously. Calculating routing caches per goal area will save calculation time and storage space because the cache can be reused more often.

Routing cache and Clusters

The multi-level routing algorithm calculates routing caches at two levels. It calculates cache for areas in a cluster and it calculates cache for cluster portals. In a map one or more clusters with areas are created. Such a cluster is a group of connected areas. Shared bounding faces and reachabilities connect the areas. The clusters are separated by cluster portals, which are areas themselves. The only way to travel from one cluster to another is through one or more cluster portals. A cluster portal always separates no more and no less than two clusters. The cache for areas in a cluster will be called area cache, and the cache for cluster portals, portal cache. The area cache stores the travel times of all areas within a cluster, including the cluster portal areas that touch the cluster, towards goal areas that are in the same cluster. The portal cache stores the travel times of all portal areas in a map, towards a goal area which can be anywhere on the map. Such a goal area can be any area from any cluster, including cluster portal areas.

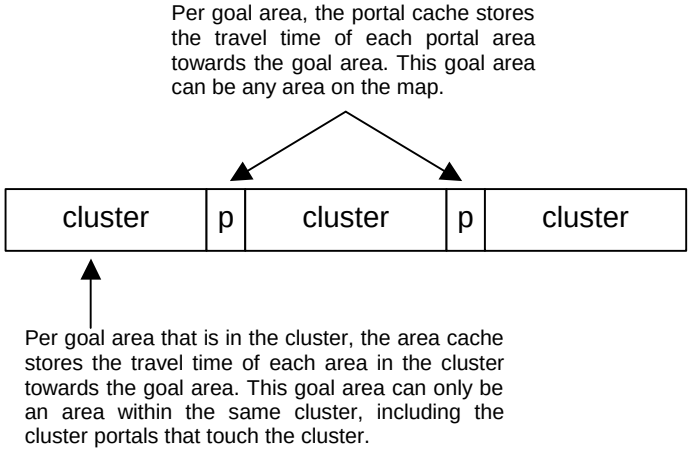


Figure 6.27: Clusters separated by portals.

Figure 6.27 shows three clusters separated by two portals both denoted with a ‘p’.

In general not all routing cache will be calculated. Routing cache will only be calculated and stored for areas the bot has had, or still has as a goal. The maximum memory requirements to store all the possible routing cache is given by the following equation:

$$M = TNA * NCP + \sum_{i=1}^{NC} \left(NA_i\right)^2 \qquad \text{Eq. 6.5.1}$$

TNA = Total Number of Areas
NCP = Number of Cluster portals
NC = Number of Clusters
NA_i = Number of Areas in cluster i

Assume cache is created for all goal areas in a map. In that case portal cache is created, which stores for each goal area, the travel times towards this goal area from all cluster portals. Added to that all area cache is created within each cluster. Within each cluster, cache is created for every goal area within the cluster. The number of possible goal areas within a cluster is equal to the number of areas within the cluster including the cluster portals that touch the cluster. For each such goal area travel times towards this goal are stored, of all areas within the same cluster.

As an example a map with a total of 100 areas is used. 10 clusters are created with 10 areas each. 9 cluster portals separate these clusters. The total number of travel times that would ever need to be stored in cache is: 100 * 9 + 10 * (10 * 10) = 1900. Now if the 100 areas within the map would all be part of one big cluster 100 * 100 = 10000 travel times would have to be stored. This makes a quite significant difference in required storage space. The less travel times that ever need to be stored, the less calculation time will also be required to calculate them.

Calculating routing caches

The area cache is calculated with a simple breadth first routing algorithm [11]. The areas are assumed to be nodes of a graph and the reachabilities the reversed links between the nodes. The breadth first algorithm starts at the goal area and uses the reversed reachability links to flood to other areas. The algorithm never floods to areas outside the cluster. The algorithm does flood into cluster portals that touch the cluster. The reachabilities store a travel time, which is the time it takes the bot to travel along the reachability. These travel times are used in the routing algorithm. The areas are assumed to be nodes of a graph, but of course the areas are not points in space. It also takes time to travel through an area. These travel times through areas are also used in the routing algorithm. For each area a small table is used with travel times from every end point of a reachability that leads towards this area, to every start point of a reachability that starts in this area and leads to another area.

The portal cache is also calculated with a breadth first routing algorithm. The cluster portal areas are assumed to be nodes of a graph, and the routes through the clusters between these portal areas, are assumed to be the links between the nodes. The goal area is also assumed to be a node in the graph. The routes from the goal area, through the cluster the goal area is in, towards portals of that cluster are also used as links in the graph. The travel times for routes through clusters can be retrieved from the area cache for those clusters.

Dijkstra's algorithm could also be used instead of the breadth first algorithm. However the areas in the Quake III Arena maps are usually not that different in size. Also the travel times stored with reachabilities between areas are not that different. Due to these properties of the Quake III Arena maps the sorting of the "nodes" in Dijkstra's algorithm usually takes up more time than the algorithm is faster by only visiting "nodes" once.

Using routing caches

Now that routing caches can be created they can be read from, and used for routing purposes. Functionality is created to retrieve the travel time from a point within a source area towards a goal area. At the same time the first reachability to be used from this source area towards the goal area can also be retrieved. Reading this data from the routing cache works as follows. In case the source area is a cluster portal area, the travel time and reachability can be read directly from the portal cache. If the source area is not a cluster portal then the travel time towards the goal is taken from each portal area that touches the cluster the source area is in. To each of these travel times, the travel time from the source area towards the specific portal area within the cluster is added. From all the added travel times the smallest one is taken. This is the travel time towards the goal in case the goal area is not in the same cluster as the source area. The first reachability to be used from the source area is retrieved accordingly. In case the goal area is in the same cluster as the source area, the travel time from source to goal within the cluster is also retrieved. Then the smallest travel time is taken from the ones through the portal areas and the one within the cluster. Here the first reachability to be used from the source area is also retrieved accordingly.
The routing cache is created on request. As soon as the cache is needed and it is not already available it is calculated. Retrieving routing information from the cache involves a loop over the portals of the cluster the source area is in and some additions. However the number of portals that enclose a cluster is usually very small, often 10 or less. This makes retrieving routing data from the cache rather fast.

Optimal clusters

It has been shown that by creating clusters in a map the maximum memory requirements for routing cache can be minimized. The question how to create clusters that are optimal remains. First of all, clusters that all have about the same number of areas will be preferred. If all clusters have the same number of areas then calculating and storing routing data will take up an equal amount of time and space for any combination of source and goal area. To minimize the calculation time and memory requirements for routing data, the equation below should be minimized. From the equation follows that the number of cluster portals should be minimized at all times.

$$M = TNA * NCP + \sum_{i=1}^{NC} \left(NA_i\right)^2 \qquad \text{Eq. 6.5.1}$$

TNA = Total Number of Areas
NCP = Number of Cluster portals
NC = Number of Clusters
NA_i = Number of Areas in cluster i

All clusters are assumed to have the same number of areas. At least X-1 cluster portals are required to separate X clusters. Equation 6.5.1 can now be written as:

$$M = TNA * (NC-1) + NC * (TNA / NC)^2 \qquad \text{Eq. 6.5.2}$$

The number of portals is the number of clusters (NC) minus one. Each cluster will have the total number of areas (TNA) divided by the number of clusters (NC) areas. Equation 6.5.2 can be written as:

$$M = TNA * NC – TNA + TNA^2 / NC \qquad \text{Eq. 6.5.3}$$

The minimum of this equation is found by setting the derivative $\frac{d}{dNC} M$ equal to zero for the variable NC.

$$TNA + TNA^2 * -1 * NC^{-2} = 0$$
$$TNA - TNA^2 / NC^2 = 0$$
$$TNA^2 / NC^2 = TNA$$
$$TNA / NC^2 = 1$$
$$TNA = NC^2$$
$$NC = \sqrt{TNA} \qquad \text{Eq. 6.5.4}$$

From the above follows that the optimal number of clusters is the square root of the total number of areas in a map. In the optimal case the total number of travel times that would ever need to be calculated and stored is approximately $2 * TNA * \sqrt{TNA}$. This grows significantly slower with the total number of areas, than TNA^2, which would be required when a conventional routing algorithm is used. For instance with 5000 areas $2 * TNA * \sqrt{TNA}$ is a little bit more than 700.000. On the other hand TNA^2 is 25 million.

Now that the optimal number of clusters is known under certain conditions the clusters still have to be created inside the map. Clusters are created by marking areas as cluster portals. A special algorithm is used to automatically mark areas as cluster portals. This algorithm uses geometric properties within the map to mark certain areas as cluster portals. Areas within door openings and windows are marked by this algorithm. These are often good cluster portals. However choosing those, the number and size of clusters are not automatically optimized. For this purpose the map designers can create cluster portals by hand. The map designers can do this by placing a special cluster portal brush in the map. These cluster portal brushes are compiled into the BSP tree like water brushes (section 6.2). The areas created from such a cluster portal brush are marked as cluster portals. By placing these cluster portal brushes the map designers can optimize the number of clusters and the number of areas in each cluster by hand.

More levels

The routing algorithm presented here uses two levels. The first level being routing between the areas within a cluster. The second level being the routing between clusters. Of course one can extend upon this idea. A third level could be added where several clusters are grouped together, and even more levels could be added. Using a routing algorithm with more levels can further reduce memory and calculation time requirements. However it also introduces more overhead when for instance reading routing data from the caches. In Quake III Arena the number of areas within maps is not that tremendous that more than two levels are needed.

6.6 Entities

Other entities in the game like other players, items, moving objects like doors, etc. provide some of the most important information for the bot. Information about these entities, like their type, their position, bounding box size etc. is communicated from the game program to the Area Awareness System. Within AAS, the information is stored for usage by the bot and the entities are linked into the areas based on their position and bounding box size. The environment sampling functionality described in section 6.3 is used to find the areas the bounding box of an entity is in. The entities are linked into the areas using a 2-dimensional linked list. This list works like the table shown in figure 6.28. A dot means the entity is in a certain area.

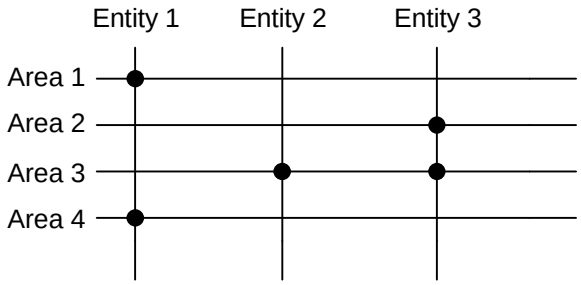


Figure 6.28: Entities linked into areas.

At the dots there is a link, which allows to easily retrieve all entities within a certain area, and to retrieve all the areas a certain entity is in. This information is useful for instance when the bot decides to retrieve a certain item on the map. The bot will need to know in which area the item entity resides in order to travel towards, and pick up the item.

7. Basic Actions

7.1 Human and Bot Input Interface

People that play Quake III Arena most often use the keyboard and mouse as input devices. The keyboard and mouse buttons are used for actions like ‘move forward’, ‘move left’, ‘jump’, ‘fire weapon’ and ‘switch weapon’, etc. The left to right mouse movement is used to turn the players view. The forward – backward movement of the mouse is often used to look up and down.
The artificial player does not use these input devices. The bot only lives inside the computer. However the bot is provided with a similar interface with the same kind of functionality. The bot will use a range of basic actions that have the same, or similar results as the input devices used by a human player. All the decisions made by the bot will eventually be translated into a sequence of basic actions. This sequence of basic actions is the bot’s input for the game.

Although these basic actions present an interface very similar to the interface provided to human players, there is a slight difference. For instance in theory a human player has no limitations on how fast he or she can change the view angles. However this is not true in practice. The hands and arms of human players have physical limits to how fast they can move, and how fast they can react to certain stimuli. The mouse itself, often used to change the view angles, has physical limits too. The basic actions do not provide these limits for the bot. These limitations will have to be explicitly implemented in the bot AI to make the bot appear more human-like.

7.2 Actions

The basic actions the bot can use as input for the game are listed below.

Attack	This action equals to a player holding down the fire button. The bot will fire the weapon the bot currently holds.
Use	The bot will use the currently held ‘holdable item’.
Respawn	This action causes the bot to respawn when the bot is dead. Human players use the fire button for this purpose.
Jump	This action makes the bot jump up.
Crouch	This will make the bot crouch. The bot will crouch while this basic action is used. Just like a human player would hold down the crouch button.
Move Up	This action makes the bot move up when swimming. This equals to holding down the jump button for human players.
Move Down	This action makes the bot move down when swimming. This equals to holding down the crouch button for human players.

Move Forward	When the bot walks this action will make the bot move in the horizontal view direction. However when under water this action makes the bot swim in the 3D direction the bot is viewing.
Move Back	When the bot walks this action will make the bot move in the opposite of the horizontal view direction. However when under water this action makes the bot swim backwards relative to the 3D direction the bot is viewing.
Move Left	This action makes the bot move sideward to the left.
Move Right	This action makes the bot move sideward to the right.
Walk	While the bot uses this action the bot will walk instead of run.
Talk	While the bot uses this action a chat icon will appear above the bot's head and the bot also won't be able to move.
Gesture	When this action is used, the visible in game model the bot uses will show a special animation of a gesture.
Move	This action is always used with two parameters: the direction of movement and the speed. The action makes the bot move in a certain direction with the specified speed. Note that unlike the common human interface the movement direction for this basic action is independent from the view angles.
View	This action will set the bot's view angles. This action uses the desired view angles as a parameter.
Select Weapon	This action sets the bot's weapon to be used. The weapon is selected with a number, which is the parameter to this action.
Command	Several commands can be typed on the console in the game. The bot can use these commands with this action, which has the command string as a parameter
Say	With this action the bot can say someone to all the other players. The message to be displayed is the parameter to this action.
Say Team	This action is similar to the Say action however now the message will only be visible to the bot's team mates.

8. Bot Characters

8.1 Characters

A human player can play the game with one or more artificial players. To make the game more enjoyable and more versatile, there is a whole range of different bot characters that play the game in their own style, and provide different challenges for the human player. There is a centralized set of characteristics for each different bot character. These characteristics are stored in the character module in the 2nd layer of the structure shown in figure 5.1. A character consists of a carefully selected set of characteristics that are applicable to the game. Of course there is no interest in characteristics that have no relation to the game, like for instance the hairstyle of the bot. The characteristics in this set are separate values, that affect how a bot behaves within the game in such a way that other players can and will notice the differences between characters.

The more characteristics that can be changed per character, the more versatile the characters can be. When more variables come into play, it can also make the different bot characters less predictable. However, simply adding many characteristics to the set does not work. A lot of characteristics interact with each other, or they depend on each other in some way. Sometimes characteristics can even contradict each other. Interaction cannot always easily be avoided. For instance there is a characteristic “jumper” and a characteristic “croucher”. These characteristics represent the tendency to jump and tendency to crouch respectively (mostly in fights to avoid projectiles). However a bot cannot jump and crouch at the same time. So if for instance the characteristic “jumper” is set to a high value this does not necessarily mean that the bot will jump all the time, because the characteristic “croucher” could also be set to a high value. This example clearly shows the interaction between two characteristics, which is sometimes inevitable. However interaction is avoided and all contradiction between characteristics is removed by choosing characteristics with clear boundaries and without overlap.

The characteristics also need to be normalized. The range of the value of a characteristic and its influence on how the bot behaves within the game, should make sense. For instance if there is a characteristic “camper” then the highest value should mean that the bot camps pretty much all the time. At the same time a very low value for this characteristic should mean the bot rarely camps. The characteristics also need to be normalized relative to each other. For instance if there is an aim accuracy characteristic for each of two weapons, then the values of these characteristics should have the same effect for both weapons. How these values scale for each weapon can be quite different depending on the kind of weapon.

When a set of characteristics is selected one should keep in mind that the perception of human players is the most important thing. What human players think about how a bot plays the game is more important than how the bot really plays the game. When all the values of characteristics, that influence how a bot plays the game, are the same for all characters, people can still have the illusion that certain bot characters play the game quite differently based on the bot’s appearance and/or name. One has to make sure the

used characteristics really do make a difference that can and will be noticed by human players.

8.2 Characteristics

The characteristics that define a Quake III Arena bot character are listed below.

Name	Name of the bot.
Gender	Gender of the bot (male, female, it – mechanical creature).
Attack skill	How skilled the bot is when attacking. > 0.0 & < 0.2 = don't move >= 0.2 & < 0.4 = only move forward/backward >= 0.4 & < 1.0 = circle strafing > 0.7 & < 1.0 = random strafe direction change > 0.3 & < 1.0 = aim at enemy during retreat
Weapon weights	File with weapon selection fuzzy logic.
View factor	Scale factor for difference between current and ideal view angle to view angle change.
View max change	Maximum view angle change per second.
Reaction time	Reaction time in seconds.
Aim accuracy	Accuracy when aiming, a value between 0 and 1 for each weapon.
Aim skill	Skill when aiming, a value between 0 and 1 for each weapon. > 0.0 & < 0.9 = aim is affected by enemy movement > 0.4 & <= 0.8 = enemy linear leading > 0.8 & <= 1.0 = enemy exact movement leading > 0.6 & <= 1.0 = splash damage by shooting nearby geometry > 0.5 & <= 1.0 = prediction shots when enemy is not visible
Chats	File with individual bot chatter.
Characters per minute	How fast the bot types.
Chat tendencies	Tendencies to use specific chats when things happen.
Croucher	Tendency to crouch.
Jumper	Tendency to jump.
Walker	Tendency to walk instead of run.
Weapon jumper	Tendency to rocket jump.
Item weights	File with item goal selection fuzzy logic.
Aggression	Aggression of the bot.
Self preservation	Self preservation of the bot.
Vengefulness	How likely the bot is to take revenge.
Camper	Tendency to camp.
Easy fragger	Tendency to go for cheap kills.
Alertness	How alert the bot is.
Fire throttle	Tendency to fire continuously instead of pausing between shots.

Most characteristics have values in the range [0, 1]. The higher the value the more the characteristic is true. The characteristics “weapon weights” and “item weight” are references to the locations where fuzzy logic is stored for situation dependent weapon preferences, and item goal selection respectively. This fuzzy logic is described in section 9. The characteristic “chats” is a reference to the location where the individual bot chatter is stored. These chats are described in section 10.
Most of the above characteristics are related to the fighting behaviour of the bot. This is not without reason. The bot is most often seen by human players during fights. As a

result the differences in behaviour between different bot characters is most noticeable when the bots are fighting.

Changing the characteristics to create a specific character can lead to interesting behaviour. First of all it is questionable if creating a perfect or extremely good bot is interesting. In general the answer to that question is no. A bot should be fun to play with. A bot should also be suitable for training purposes. A bot that is just a little bit better than the human player is often very suitable for training and practice. A perfect bot would not be any fun to play with. Human players do not want to loose the game continuously. They at least need to win every once in a while or need to know they can eventually beat the game after a lot of practice. If the bot does not leave that option it will only frustrate human players.

The impact of slightly changing characteristics can sometimes go unnoticed. At the same time small changes can sometimes drastically change the bot's behaviour. Getting all the values of the characteristics right, and in balance can be a time consuming task, which requires lots of testing. In Quake III Arena the bot characters do not change during gameplay. The different bot characters are designed to be versatile yet fixed. When a certain bot character is chosen for play, it will always have the same style of playing and will be equally skilled as any other time that same bot is chosen to play with.

9. Bot Decisions & Preferences

9.1 Fuzzy Logic

The bot uses fuzzy relations (section 3.4) to specify how much the bot wants to do, have or use something. For instance a fuzzy value is attached to how much the bot wants to have a certain item. The fuzzy relations are based on the current state of the world and the state of the bot. The current state is represented by a set of criteria or variables. For each item the bot has a fuzzy relation which shows how much the bot wants to have the item. These fuzzy relations are based on variables that represent which items the bot has in its possession, how much of each item the bot has, how much health the bot has, how much armor the bot has etc. The bot can evaluate the fuzzy relation for each item, and will want the item with the heighest fuzzy value most. In the same way the bot has a fuzzy relation for each weapon in order to choose the right weapon during combat.

Several smaller or one larger neural network could be used for the same purpose. For instance a neural network could be trained for each item. The input to this neural network is a set of variables that represent the state of the world. The output of such a neural network would then represent how much the bot wants to have the item in question. Using one larger neural network is also an option. This neural network would have all the variables that represent the state of the world as input. The output of this neural network would directly represent the item the bot wants most.
The Quake III Arena bot does not use neural networks for the purpose of item and weapon selection. Training neural networks is often time consuming. This should not be a serious problem if the networks are trained in advance before the bot enters the game. However the training process requires algorithms of higher complexity than the ones used for simple fuzzy relations. Since each bot character uses its own personalized item and weapon preferences the memory requirements to store these preferences may not be too high. The memory requirements to store neural networks are often higher than the memory needed to store the fuzzy relations, although this somewhat depends on the amount of detail with which the fuzzy relations are described. A neural network is much like a black box where some input enters at one side and some output exits at the other side. It is rather difficult to see which rules this black box implements. Although it somewhat depends on the representation used to describe fuzzy relations, it can be a lot easier to spot and adjust certain rules in such fuzzy relations. The advantage of using neural networks could lie in their power to generalize from a collection of samples. However a similar generalisation can be achieved with fuzzy relations.

9.2 Representation

The fuzzy relations that the bot uses are stored using a tree-like structure. The leaves of this tree store fuzzy values. A node in the tree links to either leaves or nodes on the next level of the tree. Each node selects one of the criteria, which is used for the evaluation of the fuzzy relation. The fuzzy relations are stored in plain text using a C-like syntax.

```
weight "name"
{
  switch( /*one of the criteria*/ )
  {
    case /*smaller than a certain value*/:
    {
      switch( /*one of the criteria*/ )
      {
        case /*smaller than a certain value*/: return /*a fuzzy value*/;
        case /*smaller than a certain value*/: return /*a fuzzy value*/;
        default: return /*a fuzzy value*/;
      }
    }
    case /*smaller than a certain value*/: return /*a fuzzy value*/;
    case /*smaller than a certain value*/: return /*a fuzzy value*/;
    default: return /*a fuzzy value*/;
  }
}
```

First of all every fuzzy relation (called a weight) has a name. Different fuzzy relations are identified by their name. C-like switch statements represent the nodes of the tree-like structure. The switch statement selects one of the criteria or variables that represent some part of the state of the world. The case keywords divide the value range of the criterion into several chunks. A division of the value range is made at the value that appears after the case keyword. The number of case keywords determines the number of divisions. A case statement 'links' to a node or leaf on the next level of the tree when the value of the criterion is below the value that appears after the case keyword. A switch always has a 'default' statement. If the value of the criterion is higher than the value that appears after the last case statement then the default keyword links to the next node or leaf in the tree. A leaf of the tree denoted by the keyword 'return' stores a fuzzy value.

Evaluation of a fuzzy relation

These fuzzy relations could be evaluated as follows. Evaluate a criterion at each node and go to the next node or leaf depending on the value of the criterion. When a leaf is reached the fuzzy value is 'returned'. However this is not the best way to evaluate the fuzzy relation. Evaluating the fuzzy relation recursively with interpolation between criterion divisions, results in a continuous range of fuzzy values. When a fuzzy relation is evaluated like that it more or less smoothly generalizes the information specified in the representation described above.

Fuzzy relation code pros and cons

There are several advantages to using the tree like structure presented here. First of all the structure is easy to understand. The fuzzy relations can also easily be deduced from situations. A snapshot at a certain moment in time can be taken, and all relevant criteria can be added into the structure with fuzzy values. The fuzzy relations can easily be created from logical reasoning and predicting common situations. The structure can also easily be modified and adjusted. In combination with the recursive evaluation of the fuzzy relation with interpolation between criterion divisions the structure generalizes and covers all situations.

The disadvantage of using the tree like structure with switch and case statements is that the structure grows rather large and becomes difficult to read when expressing some of the more complex relations.

9.3 Preferences

The bot has item weights to decide which item it wants most. Fuzzy relations as described above are used for the item weights. Every bot character has it's own individual item weights.

```
weight "holdable_teleporter"
{
  switch(INVENTORY_TELEPORTER)
  {
    case 1:
    {
      switch(INVENTORY_MEDKIT)
      {
        case 1: return 60;
        default: return 0;
      }
    }
    default: return 0;
  }
}
```

Figure 9.1: Teleporter item

The above fuzzy relation gives the fuzzy weight for the teleporter item, which is a holdable item. INVENTORY_TELEPORTER is a variable that is 1 when the player has a personal teleporter. INVENTORY_MEDKIT is a variable that is set to 1 when the player has a med kit. A player can only hold one holdable item at any time. A fuzzy weight higher than zero is only returned when the bot has no teleporter and no med kit. For each item the bot has such a fuzzy relation. The bot prefers the item with the highest weight.

The bot also has weapon weights to decide which weapon to use during combat. Fuzzy relations as described above are used for these weapon weights. Every bot character has it's own individual weapon weights.

```
weight "Lightning Gun"
{
  switch(INVENTORY_LIGHTNING)
  {
    case 1: return 0;
    default:
    {
      switch(ENEMY_HORIZONTAL_DIST)
      {
        case 768:
        {
          switch(INVENTORY_LIGHTNINGAMMO)
          {
            case 1: return 0;
            case 50: return 70;
            case 100: return 77;
            case 200: return 80;
            default: return 80;
          }
        }
        case 800: return 0;
        default: return 0;
      }
    }
  }
}
```

Figure 9.2: Lightning gun

The above fuzzy relation shows the preference a bot has for using the lightning gun during a fight. INVENTORY_LIGHTNING is a variable that is set to 1 when the bot has the lightning gun. ENEMY_HORIZONTAL_DIST is a variable that represents the distance towards the enemy. INVENTORY_LIGHTNINGAMMO is a variable that represents the amount of ammunition the bot has for the lightning gun. The lightning gun is used only when both the bot has the weapon and the enemy is within range. The range of the lightning bolt is 768 units. Based on the amount of ammunition the bot has for the lightning gun the bot will more or less prefer using the weapon. From the fuzzy relation above follows that when the bot has the lightning gun, and the enemy is within range, the fuzzy weight is defined by the function shown in figure 9.3.

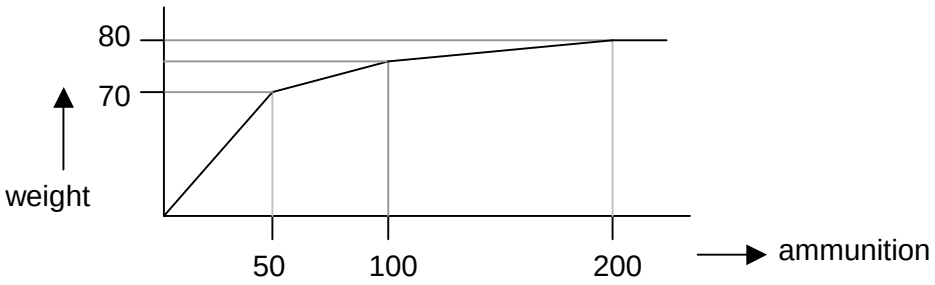


Figure 9.3: Lightning gun fuzzy weight.

For each weapon the bot has such a fuzzy relation. Each of these fuzzy relations describes how much the bot wants to use one of the available weapons during combat. The bot prefers the weapon with the highest weight.

In the above examples the fuzzy values are not within the range [0, 1]. For instance values like 70 or 80 are used. The range used for the fuzzy values has no impact on the decisions and/or preferences of the bot, as long as all the weights are in balance relative to each other.

9.4 Genetic Selection

Making sure all fuzzy relations for item and weapon preferences are in balance can be a difficult and time consuming task. Genetic selection can be used to make optimizing and balancing the fuzzy relations easier. With the genetic selection one can strive to balance the fuzzy relations in such a way that certain properties and characteristics surface.

A specific number of bots, for instance 10, all using the same fuzzy logic, fight in duals. When each bot has fought a specific number of duals the bots are ranked. These rankings can be based on the number of wins and losses but other rankings are also possible. Based on these rankings genetic selection is used to select two parents and one child. The higher a bot is ranked the higher the chance that bot will be chosen as a parent. The lower a bot is ranked the higher the chance that bot will be chosen as the child. The fuzzy logic from both parents is interbreeded. This is fairly easy because all bots started out with the same fuzzy logic and thus the tree-like structures are all the same. Averaging between the fuzzy values of the fuzzy relations from both parents can be used to interbreed. The interbreeded fuzzy relations replace the fuzzy relations from the child. The child has now new fuzzy relations. The last step in this process is mutating the fuzzy relations of the child. Slightly changing the values in the fuzzy relations with random values is an option. When the fuzzy relations of the child are mutated all bots will fight again in duals and the whole process is continued.

After repeating this process many times the fuzzy relations of the bots will tend to have specific properties and characteristics based on the rankings, which were used for the genetic selection. Repeating the whole process many times would be quite time consuming if it were not possible to scale the time in Quake III Arena. Letting the time advance much faster would make the game unplayable for human players. However this does not provide a problem for bots. The bots can easily ‘think’ as much faster as the time advances faster.

10. Bot Chats

10.1 Communication with text

Quake III Arena includes team based game types like regular teamplay and CTF. The bot needs the ability to play these game types and has to operate in teams. As a result the bot will have to communicate with other players, both human and artificial. Human players communicate with each other using text or chat messages. A Quake III Arena bot does the same. The bot has to interpret messages from others and has to create chat messages itself. Aside from the communication in a team the bot can also say things based on certain events. For instance the bot could say something like “your aim is bad” when the bot has killed an opponent.

All the chat messages the bot outputs are constructed in advance. Some variation is possible by using random or variable strings as part of a message. The interpretation of chat messages from other players is relatively simple. The sentences are not decomposed into verbs, nouns, adjectives etc. The interpretation is based on keywords found in the chat messages. A chat message can also be compared to known templates in order to give meaning to the message.
Complex parsing and decomposition usually does not work very well in a fast pace game like Quake III Arena. There is often not enough time for such complex analysis of sentences. The parsing would also often fail because people have to type the chat messages rather quickly, which causes many mistakes, errors and misspelled words. The usage of slang in chat messages is also not uncommon.

10.2 Interpreting text sentences

Synonyms

The synonyms are used to unify chat messages from others, before further processing is done. They are also used to add variation when the bot constructs a chat message. The bot uses context dependent synonyms. Every context has a special flag.
A context with synonyms is stored in a text file as follows:

```
context flag
{
  [("word", X), ("a synonym of word", Y), ...]
  ...
}
```

The X and Y are values in the range [0-1]. These values are chances the synonyms are used in chat messages the bot outputs. A chance of zero means the bot will never use the synonym.

A few examples:

```
CONTEXT_NEARBYITEM
{
  [("Heavy Armor", 0), ("red armor", 0), ("Heavy Armour", 0), ("red armour", 0), ("ra", 0)]
}

CONTEXT_NORMAL
{
  [("do not", 1), ("don't", 1), ("dont", 0)]
  [("checkpoint", 1), ("check-point", 1), ("cp", 0)]
}
```

When the bot unifies a chat message it replaces all synonyms with the first one listed.

Match templates

The bot uses the match templates to interpret and “understand” chat messages from other players. The bot tries to match a message it receives with one of the match templates. A match template is stored as follows:

template = (id, type flags);

Fixed strings and variables alternate in a match template, separated by commas. For instance: "you are ", 0, " don't you think?" The fixed strings are placed between double quotes. The 0 is the index of a variable. A maximum of 8 variables is allowed (index in the range [0-7]). It is not allowed to have two consecutive variables.
The bot will try to find a match with one of several fixed strings in the template when they are separated by the **|** token. For instance: "you"|"we", " are ", 0, " don't you think?" will match to both "you are crazy don't you think?" and "we are crazy don't you think?". The bot will try to match the strings separated by the **|** token in the order they are listed. An empty string is also allowed with the **|** token. However the empty string always needs to be the last one in a sequence of strings separated by **|** tokens, because the empty string will always match. For instance: "I am the ", "team "|"", "leader" will match to both "I am the team leader" and "I am the leader".
The id is used to identify the message. The type flags specify certain properties of the message. The bot uses this id and the flags to give meaning to the message.

The bot uses context dependent match templates. Every context has a context flag and several match templates can be grouped within a context.

```
context flag
{
  ...
}
```

An example:

```
MTCONTEXT_INITIALTEAMCHAT
{
  "(", NETNAME, "): ", ADDRESSEE, " defend ", KEYAREA,  “ for “, TIME = (MSG_DEFEND, 0);
  ...
}
```

The names NETNAME, ADRESSEE, KEYAREA, TIME and MSG_DEFEND are macros for numbers. The bot uses the match templates in the order they are stored. The first match found is used.

10.3 Initiating chats and Eliza chats

Just like human players the bots will sometimes say things when the environment changes. For instance when killed, the bot could say something like “nice shot” to the enemy. The bot can also reply to things other players say. For instance when someone says to the bot “you are good” the bot could reply with “no I’m not that good”.

Initial chats

The initial chats are used by the bot to initiate a chat when something in the environment changes or the bot just feels like chatting. The bot can also use them to respond to something a team mate says. Each bot has a personal set of initial chat lines. In the characteristics of a bot there is a reference to a location where the initial chats for that bot are stored. The initial chats for a bot are stored as follows:

```
chat
{
  type "type name"
  {
    "initial chat message";
    "another initial chat message";
    ...
  }
  type "type name"
  {
  }
  ...
}
```

The bot can select messages based on the type name. One of the messages listed for a specific type is chosen at random.

Reply chats

The bot uses the “reply chats” to reply to chat messages from other players. These reply chats work very similar to the Eliza chat program [web. 3]. This chat program was named after Eliza Doolittle and was created by MIT scientist Joseph Weizenbaum. Its mission was to attempt to replicate the conversation between a psychoanalyst and a patient. Eliza talks you into giving her your deepest feelings. She will not remember it in the next sentence and keeps no recollection of anything you say. However, she has a personality like no other program. The bots use a similar system with chat messages that are applicable to the game.

All bots use the same reply chats because the number of reply chats tends to grow quite large. Creating different reply chats for each bot character would be too much work and also take up to much memory.

Each reply chat has keys, a priority and several reply chat lines.

```
[key1, key2, ...] = priority
{
  "reply chat message";
  "another reply chat message";
  ...
}
```

Possible keys:

name	key is true when the name of the bot is found in the chat message
female	key is true when the bot is female
male	key is true when the bot is male
It	key is true when the bot is not male nor female
<"","","">	key is true when the bot has one of these names, any number of names can be listed between the < and >
""	key is true when the string between the quotes is found in the message to reply to

( )	key is true when the chat message to reply to matches the template between the ( ) The match templates work in exactly the same way as the match templates described above. The matched variables can be used in the reply chat messages. For example:
	
	[("I'm not ", 0)] = 4
	{
	"yes you are ", 0;
	}

The priority is a value that is relative to the priorities of other reply chats. The bot evaluates the keys to find out if the reply chat can be used to reply to a certain chat message from another player. A key can be preceded by a **&** which means that the key has to be true. When a key is preceded by a **!** the key must be false. When all the keys with the prefix **&** are true, and all the keys with the prefix **!** are not true, and there is at least one key without prefix true, then the bot may use the reply chat. When several reply chats can be used to reply to a message the one with the highest priority is chosen.

Reply chat examples:

```
["hate you", !"not"] = 7
{
  "why do you hate me";
  "there's no reason for you to hate me";
}
```

Bots can use the above reply chat when someone types the message "I hate you". However if a player types the message "I don't hate you" the bot cannot use the above reply chat because of the key "not" with the prefix !. (Note: "don't" will be replaced by "do not" because of the contraction synonyms in the synonym list.)

```
["love you", !"not", &female] = 6
{
  "am I the only woman you love?";
  "I love you to";
}
```

Only female bots can use the above reply chat.

```
["camper", !"not", &<"Grunt", "Stripe">] = 6
{
  "I love camping";
  "I'm the king of all camp grounds";
  "camping with the rocket launcher is what Graeme told me to do";
  "so?.. you got a problem with campers?";
}
```

Only the bots Grunt and Stripe can use the above reply chat.

Bad reply chats

Unfortunately the described representation allows the implementation of reply chats that do not work well or do not work not at all. The reply chats that are troublesome are listed here.

```
[&"looser", !"not", &female] = 6
{
  "I'm not a looser";
}
```

All the keys in the above reply chat have either the **!** or **&** prefix. No bot will never use this reply chat because at least one key without prefix must be true.

```
[("I hate ", 0), "hate you"] = 7
{
  "are you sure you hate ", 0;
}
```

In the above reply chat the variable 0 does not have to be valid when the bot tries to output the message "are you sure you hate ", 0; because the key "hate you" could be the only key that is true. For instance only the key "hate you" is true when the bot tries to reply to the message "we hate you".

```
[("get lost"), &name] = 7
{
  "I never get lost";
}
```

The above reply chat will never be used because there is no room for the name of the bot in the match template. However the name of the bot is required to be in the chat message to reply to because of the key &name.

```
[("do not say ", 0, " want to ", 1), !"not"] = 7
{
  "but ", 0, " really want to ", 1;
}
```

The above reply chat will never be used because "not" is a sub-string of "do not say ".

```
["allowed"] = 7
{
  "everything is allowed.";
}

["not allowed"] = 5
{
  "why isn't that allowed?";
}
```

If both of the above reply chats exist then the bot will never reply with "why isn't that allowed". If the bot for instance wants to reply to the message "cheating is not allowed" then the bot figures out it can use both of the above reply chats according to the keys. The message "cheating is not allowed" has the sub-string "not allowed" and also the sub-string "allowed". Because the keys allow the bot to use both reply chats the priority is used to select one of them. The priority of the reply chat with the key "allowed" is higher and will be used. Changing the priorities will fix this problem.

Random strings

The random strings are used in the initial and reply chat messages to add variation. A set of random strings is stored as follows:

rndname = {"first random string"; "second random string"; ...}

A set of random strings is used in a chat message as follows:
" part of the chat message ", rndname, " part of the chat message ";
Multiple references to random strings can be used in a chat messages. Multiple references to random strings separated by semi colons can also be used directly after each other.
A set of random strings may also contain references to other sets of random strings.

```
ferrari = {"F40"; "F50"}
BMW = {"BMW Z3"; "BMW Z8"}
cars = {ferrari; BMW; "F1 Mc Laren"}
```

Of course recursion has to be avoided at all time.

Chat messages

Both the initial chats and reply chats store chat messages. Random strings can be used in these chat messages to add some variation. Variable strings from match templates in the reply chats, or specified in code for initial chats, can also be used in chat message. The tilde character ~ can be placed directly in front of a word in a chat message to make sure that word will not be replaced by a synonym. All tilde characters ~ are removed from the chat messages before a bot outputs them. Colors can also be used in chat

messages. A color is specified with the **^** character followed by a number in the range 0-7. Each number specifies a different color. All text after the color specification will be shown in that color. The color specification itself is not shown in the chat message visible to the players.

10.4 Chat reasoning

The pseudo code below shows how the different parts of the chat system are used. The reasoning shown here takes place in the 3rd layer of the structure in figure 4.1.

```
if environmental change then
  if bot wants to chat then
    choose initial chat
    use random strings in chat message
    replace synonyms in chat message to add variation
    output chat message
  endif
endif
```

There are quite a few environmental changes that the bot might respond to with a chat message. For different events the bot has sets of chat messages with a different type name stored in the initial chats. The bot can often use several variables in the chat messages denoted with a number. As value these variables can for instance have the name of an opponent, the name of a map or the name of a weapon used by the bot or an opponent in a fight.

In Quake III Arena there are quite a few environmental changes that might trigger the bot to say something. The bot can for instance initiate a chat when entering or exiting the game. When the bot is already in the game and a new map starts the bot can also decide to chat. The bot can also initiate a chat when a map ends because either the bot itself or someone else wins. The bot could say something like “good game!”. When a bot is hit by an enemy projectile it might want to respond with a chat message. When the bot was typing a message at the time, the bot could say something about it. For instance “hey I’m trying to say something”. When the bot dies a whole range of different chat messages can be used. The chat message is chosen based on the cause of death. When the bot commits suicide it will say something different than when an enemy killed the bot with for instance the gauntlet. There are different chat messages for when the bot drowns, dies in lava or fell to its death. In case the bot is killed by an enemy the bot can also choose to say something positive or the bot might want to insult the enemy. Different chat messages can be used based on the death circumstances of an enemy that dies in a fight with the bot. The bot could try to insult and for instance say “you’re a bad looser” or the bot can praise the enemy for putting up a fair fight. The bot can also try to make conversation and decide to say something without having a direct cause from within the game. In this case the bot could just say something to one of the opponents either trying to insult or praise the enemy.

When something happens the bot would like to respond to, or the bot just feels like saying something, the bot will first check if it is in a good position in the game world to chat. It is for instance not wise to start chatting in the middle of an intense fight.

The pseudo code below shows how the bot deals with incoming messages.

```
if bot receives a message then
  replace synonyms in the message
  interpret message using match templates
  if match is found then
    perform action
  else
    if messages is a chat message from another player then
      if bot wants to reply to this message then
        find a reply chat
        use random strings in chat message
        replace synonyms in chat message to add variation
        output chat message
      endif
    endif
  endif
endif
```

When the bot receives a message it might decide to respond to the message. First the bot tries to “understand” the message. The message could be from a team mate asking or ordering the bot to do something. Another player could also try to provide the bot with information of some kind. The match templates are used to make sense out of such incoming messages. When a matching template is found the bot will do something with the provided information or try to accomplish what is being ordered by a team mate.

In case the incoming message is a chat message from someone else and the bot cannot make any sense out of it, the bot can decide to use the Eliza like reply chats to say something. The bot goes through all the reply chats matching the keys to the chat message, and chooses the one most suitable for reply.

11. Bot Goals

11.1 Ingame goals

Winning the game is for a bot the most important goal, as it usually is for human players. Depending on the game type a bot wants to win the game either as an individual or as player operating in a team.
In deathmatch mode winning is achieved by having the highest frag count (number of killed opponents) at the end of the game. The game ends when a player reaches the frag limit or when the time limit is hit. The bot will strive to be the player with the highest frag count at the end of the game. In regular teamplay the team with the highest accumulated frag count wins. In the capture the flag game mode the team with the most flag captures wins. In team games the bot will try to help its team towards a victory.

To win the game the bot uses a lot of sub-goals during gameplay. In deathmatch mode and of course also during fights in team games, the bot will try to kill the enemy or the direct opponent. The bot will have to aim and shoot at the enemy. On the other hand the bot will also try to stay alive during such battles. The bot will try to dodge enemy projectiles and missiles. During fights but also when no enemies are nearby, the bot will try to gain strength by picking up items, weapons and powerups.

There are different types of sub-goals, which the bot uses to eventually win the game as an individual or in a team. First of all a distinction is made between short term goals and long term goals. Short term goals are goals the bot achieves while going for, or optimizing for a long term goal. For instance the bot picks up an item, which is a short term goal, while chasing the enemy, which is the long term goal at that point. Basically the bot has winning the game as goal. Several long term goals are used during gameplay to achieve this goal. While trying to achieve the long term goals the bot uses several short term goals either to achieve the long term goal or because they can easily be accomplished on the way. All short term goals and long term goals always involve a position or location in the environment. For instance when the bot tries to retrieve a certain item the bot will navigate towards the location of that item. The bot will continuously evaluate the status on achieving the goals it has set out for. For instance if the bot wants to pick up an item, it will continue moving towards the item until the bot can see the item has been picked up, because it is no longer there. The bot also keeps track of the time it is taking trying to achieve a goal. If the bot is taking too long it will often abandon the goal and decide to do something else.

11.2 Short term goals

While trying to achieve a long term goal the bot goes for several short term goals along the route. These short term goals are usually nearby items, weapons or powerups that the bot can easily pick up without diverting too much from the path towards the long term goal. Going for air is also considered a short term goal when swimming towards a long term goal.

11.3 Long term goals

There are a variety of long term goals the bot uses in order to try to win the game. The most common long term goals are items, weapons and powerups the bot wants to have. The bot uses fuzzy relations as described in section 9 to decide which item(s) it wants most.

Figure 11.1: Two items which can be goals

When the bot ends up in a fight it can go for several long term goals. First of all the bot can go for killing the enemy in a direct fight. The bot will aim and shoot at the enemy and try to avoid projectiles and missiles from the enemy. When the bot does not feel fit enough to fight the bot might decide to retreat. Going for an item that leads away from the enemy is the long term goal in this case. When the enemy runs away the bot can decide to chase the enemy. In this case the enemy is the long term goal. When the enemy goes out of sight the bot will run towards the last seen position of the enemy hoping to see the enemy from there.

Figure 11.2: A camping bot

The bot can also decide to stay at a specific location for a while, either waiting for an item to respawn or because it is a spot from where the bot can easily take out enemies. This kind of behaviour is often called camping. In this case the long term goal is to stay near a position that gives the bot some advantage.

In team games the bot can have several additional long term goals. Usually a bot is ordered by the team leader to go for these long term goals. One of the things a bot can do is helping out a team mate. The bot will go towards the team mate and help fighting the enemies. The bot can also accompany a team mate. The bot will follow the team mate around and will help eliminating enemies encountered on the way. A bot can also be ordered to defend a key area in the environment. Usually this key area involves an item or for instance the base flag in CTF. Bots can also patrol a certain area. In this case the bot is usually ordered to patrol along certain items.

There are also some special long term goals only applicable to a capture the flag game. In CTF the bot can try to steal the enemy flag. The enemy flag would be the long term goal in this case. As soon as the bot has picked up the enemy flag it will rush back to the base. When the enemy has stolen the flag of the bot's team, the bot can have returning the flag as a long term goal.

Figure 11.3: A flag in a CTF game

12. Bot Navigation

The Area Awareness System provides all the information the bot needs to navigate through an arbitrary environment in Quake III Arena. The bot uses frame based thinking and the movement AI code runs every frame. The basic actions are used to create input for the game in order to navigate through the environment. The Area Awareness System also provides all the information required by the bot to find a path through the environment. The bot uses two types of navigation. These are navigation in a specific direction and navigation towards a goal.

12.1 Moving towards a goal

A goal towards which a bot moves always has to be in an area of AAS. The bot is also always in one of the areas of AAS itself. When the bot is in the same area as the goal the navigation is very easy due to the basic navigation property of AAS. The bot will just navigate along a straight line towards the goal. In case the bot is not in the same area as the goal, the routing system can be used to calculate the next area the bot needs to travel to in order to reach the goal. The routing system can provide the reachability towards this next area directly. However the travel times towards the goal of adjacent areas will be used to find the next area the bot needs to travel to. The reachabilities of the area the bot is in are used to find adjacent areas. The travel time towards the goal of those areas is retrieved from the routing system. The area with the smallest travel time towards the goal is chosen as the next area. The reachabilities provided by the routing system are not used directly, because the bot can decide to avoid certain areas. These areas can then be excluded when the bot decides which area it wants to travel to next. When the next area to travel to is known, the reachability towards this area is also known. The bot will use such reachabilities to travel towards the goal. For each reachability type the bot has specialized movement AI code to travel along the reachability.

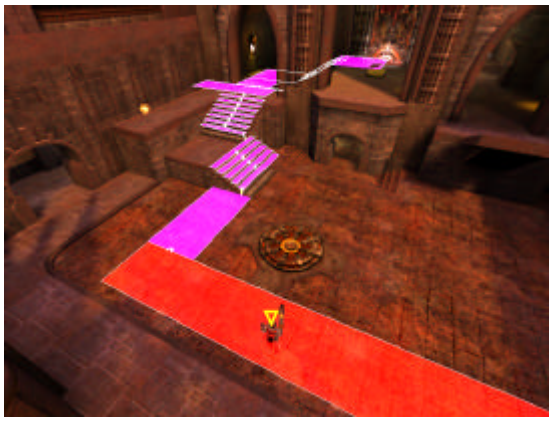


Figure 12.1: Route through areas (only area ground faces are shown).

Each movement frame the bot first checks if it is standing on top of an elevator or bobbing platform. If this is the case the bot will try to continue with the specialized AI

code for the reachability the bot was already using for this moving object. In case the bot accidentally ended up on the moving object the bot will find a reachability for it and use the specialized code for that reachability. When the bot is not standing on a moving object the bot checks if it is air-borne. For each reachability type there is specialized AI code for when the bot is air-borne and specialized AI code for when the bot is standing on the ground or swimming. Specialized code is used for when the bot is air-borne because the control over movement is very limited when the bot is flying through the air. When the bot is not air-borne the bot will continue using the reachability from the last movement frame unless the bot changed areas since. The area the bot is in, is found using the environment sampling functionality described in section 6.3. In case the bot is in a new area the bot will find a new reachability towards the next area on the route towards the goal. The bot then continues with that new reachability. The bot moves from area to area until the bot is in the same area as the goal. From there the navigation has minimal complexity.

12.2 Moving in a direction

The bot can also just move in a certain direction instead of towards a goal. The bot does not use the reachabilities or the routing system when moving in a specific direction. Only the environment sampling functionality is used.
When swimming the bot can move in the specified direction without any problems until a wall or obstacle is hit. While the bot is walking there are more different obstacles that need to be handled. When the bot runs into a barrier the bot will verify if it can jump onto the barrier. If this is the case the bot will jump onto the barrier. When there is no barrier the bot checks for gaps in the floor the bot might fall into. The bot will try to jump over gaps that are found. Before trying to jump, the jump is predicted to find out where the bot will end up. In case the bot would end up in lava, slime or a death pit the bot will not jump. When there are no barriers to jump onto and no gaps to jump over the bot will try to move in the specified direction. If the bot runs into a wall the movement will fail.
The bot will mostly use this simple navigation AI code to move in specific direction during fights, for instance to avoid enemy projectiles.

13. Bot Fighting

A bot is most often seen during direct fights. As a result the fighting behaviour of the bot is rather important and a lot of settings of bot characters apply to this fighting behaviour.

13.1 Acquiring an enemy

Before the bot ends up in a fight it has to acquire an enemy. When an enemy saw the bot first and initiates a fight the bot will quickly look around to find from where the enemy is shooting. The bot will most likely return fire and might want to look for cover. When there are no enemies initiating a fight the bot will continuously look out for danger. Information is available to the bot about the positions of all potential visible players around. However a human player has only a limited view. The field of vision is usually limited to only 90 degrees. This view limitation has to be explicitly implemented for the bot because it does not have this limitation by default. With this limitation the bot will also only "see" enemies within a 90 degrees field of vision. Aside from the limited view cone, the visibility of enemies can also be reduced by fog.

The bot, the enemy or both can be standing in a foggy area. Enemies might go unnoticed to the bot because of limited visibility due to fog. When an enemy is very far away and the appearance of the enemy only covers a small percentage of the bot's view the enemy might also go unnoticed. In this case seeing and perhaps engaging the enemy will depend on how alert the bot is. When an enemy is wearing the invisibility powerup the bot will not see nor engage the opponent. However enemy fire will make both the presence and the position of the enemy visible to the bot.

Figure 13.1: Enemy in fog

When a bot sees an enemy it is not always a good idea to initiate a fight. When the bot is low on health or does not have any powerful weapons it might not want to engage the enemy. Especially when the enemy is not facing and has most likely not yet spotted the bot it is often smart to avoid a battle. The bot will want to stay outside the enemies view and go for some nearby items, weapons or powerups before entering a fight.

13.2 Using weapons

The bot is most often seen by human players in a battle. As a result a lot of the characteristics also apply to how the bot fights in battles. There are several things very important in a fight like choosing the right weapon, aiming the weapon at the enemy, taking the right position and avoiding enemy projectiles. Using weapons, both selecting a weapon and aiming the weapon at an enemy will be looked into first.

Selecting weapon

The bot uses fuzzy logic as described in section 9 to store situation dependent weapon preferences. The preference is often based on the relative power of the weapon. Some weapons are more powerful than others and are more easily used to kill the enemy. The bot can also prefer certain weapons based on personal taste. However not all weapons can or should be used in certain situations. For instance if the enemy is out of range for the lightning bolt to hit the enemy, it is not very useful to use the lightning gun.

Aiming

When the bot has selected a weapon it still has to aim at the enemy before shooting. There are two characteristics that affect the aim of the bot. These are the aim accuracy and the aim skill. The bot selects a spot on or near the enemy to aim at. The accuracy with which the bot chooses this spot depends on the aim accuracy characteristic. Depending on the aim skill characteristic the bot will be more or less skilled in selecting a spot to aim at in order to hit or damage the enemy. When the bot is using an instant hit weapon like the rail gun then the spot to aim at is always right on the enemy. The aim skill characteristic has no influence in that case. However the aim skill characteristic is used when the bot uses a weapon that fires projectiles that travel at a lower speed and do not instantly hit. Most often the enemy will not be standing still at one position during a fight. The enemy could be moving around a lot to avoid projectiles. The enemy could also be running away, or towards an item. When the enemy is moving a lot the bot should not fire slow projectiles at where the enemy is at the time of firing. If the bot would fire at the current position of the enemy, then the enemy is most likely somewhere else by the time the projectile arrives at that position. If the bot is not very skilled it will not take enemy movement into account at all when firing. When the bot is a little bit more skilled the bot will linearly extrapolate the position of the enemy. The bot will then aim at the extrapolated position. The surrounding geometry and the path the enemy is likely following, are not taken into account. The very skilled bots will predict the enemy movement much more accurately. These bots take the physics and surrounding geometry into account to predict where to aim to get the projectile to hit the enemy.

Aside from taking the visible movement of the enemy into account the bot can also take certain aspects of weapons into account. For instance projectiles that inflict radial damage when exploding on impact can be aimed at nearby geometry. Even when the enemy tries to avoid the projectile the enemy might still get damaged from the nearby explosion.

When the enemy goes out of sight during a fight the skilled bots will predict where the enemy will show up again. The bot will shoot projectiles that inflict splash damage when exploding on impact at the position where the enemy is likely to show up. The bot uses the reachabilities and routing from AAS to predict where the enemy is likely to show up. The bot always assumes the enemy is coming back into view using the shortest path towards the bot. This shortest path from the last seen enemy position towards the bot is predicted. The bot will shoot projectiles that inflict splash damage at the first visible position along this path. This works especially well when the bot is retreating and the enemy is chasing.

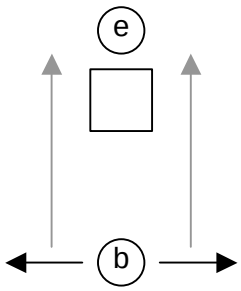


Figure 13.2: Shooting projectiles at both sides of a pillar.

Sometimes an enemy hides behind a pillar or behind a box as illustrated in the top down view in figure 13.1. The circle with the ‘e’ is the enemy. The square is the pillar or box and the circle with the ‘b’ is the bot. It is often a good idea to shoot projectiles that inflicts splash damage at both sides of the box or pillar. By doing that the enemy cannot go either way and will always get damaged by the splash damage of the projectile. The Quake III Arena bot does not know to do this. However the bot is often moving from side to side and the bot predicts where the enemy is likely to show up. This predicted position will change when the bot moves from side to side and as a result the bot will fire projectiles that inflict splash damage at both sides of the pillar or box.

13.3 Movement

To avoid enemy projectiles and to make aiming more difficult for the enemy, the bot can perform a whole range of evasive and confusing moves. Depending on the attack skill characteristic the bot will be more or less skilled in performing such moves during combat. Bots that are not very skilled will not move at all during a fight. They will be easy targets for the enemy. The somewhat more skilled bots will move back and forth towards the enemy. This makes it a little bit tougher for the enemy to aim but the bot will still be quite an easy target. When the bot is even more skilled the bot will “circle strafe” around the enemy. The bot moves sideward in one direction circling around the enemy. This makes it significantly harder for the enemy to aim at the bot. The most skilled bots will also change the strafe direction randomly while circling around the enemy.
While moving back and forth or circling around the enemy the bot can jump to make it even harder for the enemy to hit, or the bot can crouch to avoid projectiles. Depending on the ‘jumper’ and ‘croucher’ characteristics the bot will have more or less the tendency to jump or crouch respectively.

The bot will also try to take a good position based on the weapon the bot is holding. Certain weapons are most effective at a certain range from the enemy. For instance the shotgun does most damage at close range. On the other hand firing a rocket launcher close to the enemy is not very wise because the explosion of the rocket would also damage the bot.
The bot can also take positions that provide an advantage in general. Usually higher positions provide an advantage to attack an enemy. It is easier to throw projectiles down at the enemy.

14. Obstacles and puzzles

14.1 Obstacles

With the environmental structures available in Quake III Arena a vast range of different obstacles and puzzles can be implemented in the game environments. For instance doors or bars, that lead to certain items, that only open when pushing a button or activating a trigger. Quake III Arena is mostly a fighting game and thus only few obstacles and puzzles are implemented. However the Quake III Arena bot has AI for the few maps that use buttons and triggers.

Due to the nature of AAS only dynamic objects in the Quake III Arena environment have to be considered as obstacles. All “fixed obstacles” are compiled into the AAS data structure and need no special handling. The dynamic objects are entities within the environment that can in some way move or change positions. These entities include other players, doors, elevators, bridges etc. The presence or absence of such objects within the areas of AAS can be an obstacle to the bot.

The movement code in the second layer checks if the bot bumps into obstacles. A simple trace in the movement direction reveals any objects the bot would walk into. The movement code will give feedback to higher layers about any such obstacles. From there the bot can do last minute handling of these obstacles. When the bot runs into a player it can try to navigate around the player. The bot can try to do the same with a door, however if the door does not open when approached, it often has to be opened with a button or some kind of trigger. Walking into obstacles like this and then trying to figure out what to do, works in a lot of cases. However there are situations where it does not work and in general it does not make for very intelligent behaviour, when trying to work with doors and such that are activated by buttons and/or triggers. It just does not look very intelligent to first (literally) walk into a door and then go for the button or trigger to open it.

14.2 Solving Puzzles

The Quake III Arena bot has AI that does not only allow for solving more complex puzzles it also makes the bot look more intelligent. The AI used for getting around obstacles and/or solving puzzles will be explained with an example. Figure 14.1 shows the top down view of a very small map. This is not an actual map available in the game, but this test map gives a fairly clear view of the kind of puzzles the bot can solve. The picture was taken from Q3Radiant, the editor used to create and edit the Quake III Arena maps.

The red box is where the bot starts. The blue box is an item the bot would like to retrieve. The map contains several walls and bars that block the bot's path.

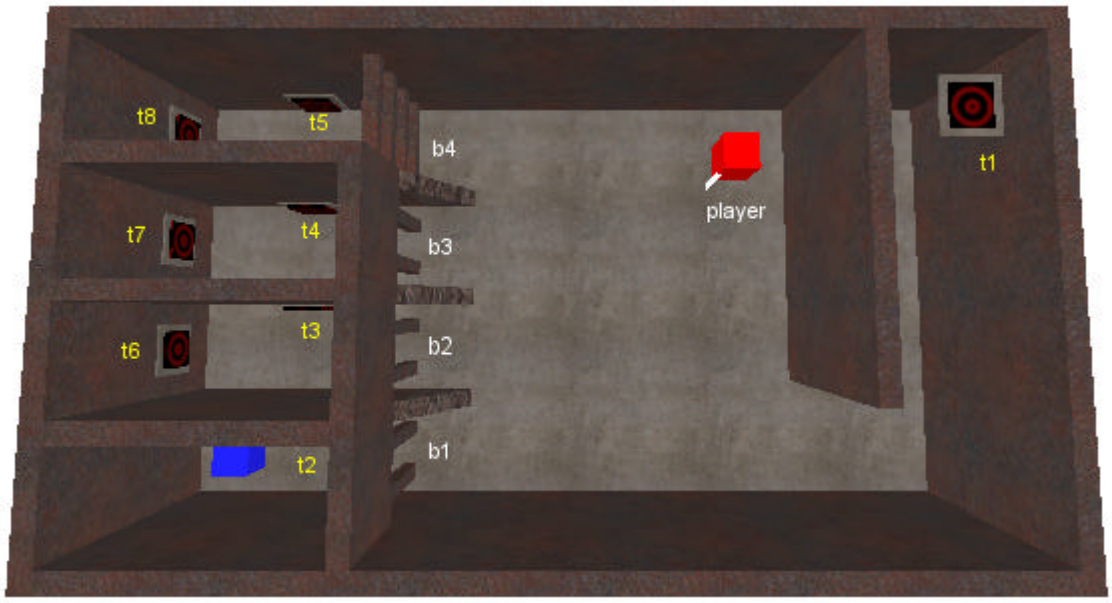


Figure 14.1: Top down view of a puzzle.

b1, b2 and b3 are vertical bars. b4 is a set of horizontal bars. t1 through t8 are buttons that trigger the bars to open. t1 is a shootable button at the ceiling. t2 through t8 are buttons positioned at walls that can be pushed. Pushing either t1 or t5 opens the horizontal bars b4. t2 and t6 open the bars b1. t3 and t7 open the bars b2. And finally t4 and t8 open the bars b3. All bars close automatically 4 seconds after being opened.

In this particular puzzle the bot has to retrieve the item behind the bars b1. The bars are not compiled into AAS as solid objects so the bot first assumes the bars are not there. The bot will walk towards the item and bump into the bars b1. The bot has the ability to find the buttons it can push or trigger in order to open the bars or activate other objects. Inside the map the button and the door are linked with a target name. The bot can look up this link to find the button(s) and/or trigger(s) that need(s) to be activated. A human being will have to learn the relation between certain buttons, and what they activate, by playing the game and trial and error. The bot could do the same but that is not a very interesting approach. As soon as the bot has learned the relations, it has the same knowledge that it now retrieves directly from the map.
In this case either of two buttons can be activated to open the bars b1, which are the buttons t2 and t6. However button t2 is behind the bars b1 itself just like the item. The bot will need to figure out it cannot go for that button. This can be accomplished by disabling the area(s) the bars b1 are in for routing. If the areas the bars b1 are in are disabled and cannot be used for routing the bot will not find a route towards button b1 and will know it cannot reach that button. There is still a small problem however. There does not have to be a set of areas that exactly contain the bars b1. For instance there could be an area that contains the bars but also stretches out in front of the bars. While standing in front of the bars the bot could be standing in that same area. If that area would be disabled for routing purposes, the bot would not be able to travel anywhere. So before any areas are disabled for routing a set of areas that tightly contain the bars has to be found.

This is not all that difficult. Just like water, lava and slime are compiled into AAS, the bars can also be compiled into AAS. New areas will be created that define exactly the volume the bars occupy while closed. If these areas are flagged as containing bars or a moving object then it will be even easier for the bot to find these areas and disable them for routing. When the areas are disabled for routing this also means all routing cache that in some way depends on these areas, has to be deleted because it is no longer valid.

Now the bot can figure out it has to activate button t6 instead of button t2. The bot will now try to activate button t6. Going towards button t6, the bot will of course run into the bars b2. The bars b2 are again opened by pushing either of two buttons. These bars can be handled in the same way the bars b1 are handled. When the bot continues like this the bot will at some point run into the bars b4. To open these bars the bot will have to walk towards a position where it can shoot the button t1, which is positioned at the ceiling. Going towards button t1 the bot will find no obstacles and the button can be activated without any problems. The bars b4 are opened. From here the bot can continue with the goal it had set out for, which was retrieving the item. If the bot has no memory it would go straight back to the item and run into the bars b1 again. Then the bot will figure out again it has to activate button t6 and go on like this. Eventually the bot reaches the bars b4 again. Even though the bot had opened these bars, by the time the bot reaches them, they have already closed again because they stayed open for only 4 seconds. If the bot continues like this, it will loop forever and never retrieve the item. The bot will need some kind of memory that stacks up the goals the bot went for, so it can complete them in reversed order. The Quake III Arena bot uses a goal stack for this purpose. Every time the bot figures out to activate a new button (which is a new sub goal) the bot will push this button goal onto the stack. Whenever the bot activates a button, the button goal will be popped from the stack. The bot then continues with the goal that is at the top of the stack. The last in, first out (LIFO) working of the stack makes sure the bot will activate the buttons in reversed order as soon as it is able to activate one of them.

At this point the bot will be able to solve the puzzle but still the bot's behaviour does not look very intelligent. Walking into the bars, and then at the last minute figuring out how to open them does not look very good, especially when the bot will always use this method. To make the bot look more intelligent the bot could predict its route or part of its route and try to find obstacles before literally running into them. The fact that the bars are compiled into AAS, and the created areas are marked as containing bars, makes finding obstacles along a predicted route a lot easier. The route towards the goal of the bot can easily be predicted using the reachabilities found with the routing algorithm, and stored in the routing cache. The bot can check for obstacles as soon as the predicted route passes through an area, which is flagged to contain bars, a door etc. As soon as such an area is found, the bot can figure out if the area really contains an obstacle, and if that obstacle needs to be removed by activating a button or trigger.

15. AI network

15.1 The network

The central “brain” of the bot is a network with special nodes for different situations and different goals. This AI network resides at the 3rd layer of the structure shown in section 5. All the other AI sub-systems are used from or within this network. This “brain” of the bot is very much like a finite state machine modeled as a network of nodes with conditional links between the nodes. The bot can only be at one node at any time. Every think frame the bot goes through this network until the node best suitable for the bot’s current situation is found. There is always exactly one node best suitable for the current situation, and the bot changes nodes until it finds this specific node. Each node within the network is optimized for a specific range of goals or sub-goals. The network also has nodes which allow the bot to do pressing tasks while going for a long term goal. Each node has a procedure with production rules (if-then-else) for the reasoning and decision making of the bot. Such a procedure also implements the conditional jumps to other nodes to make sure the best node is found for each situation. If a certain node is the one best suitable for the bot’s current situation then the procedure for that node returns true. Otherwise the procedure returns false.

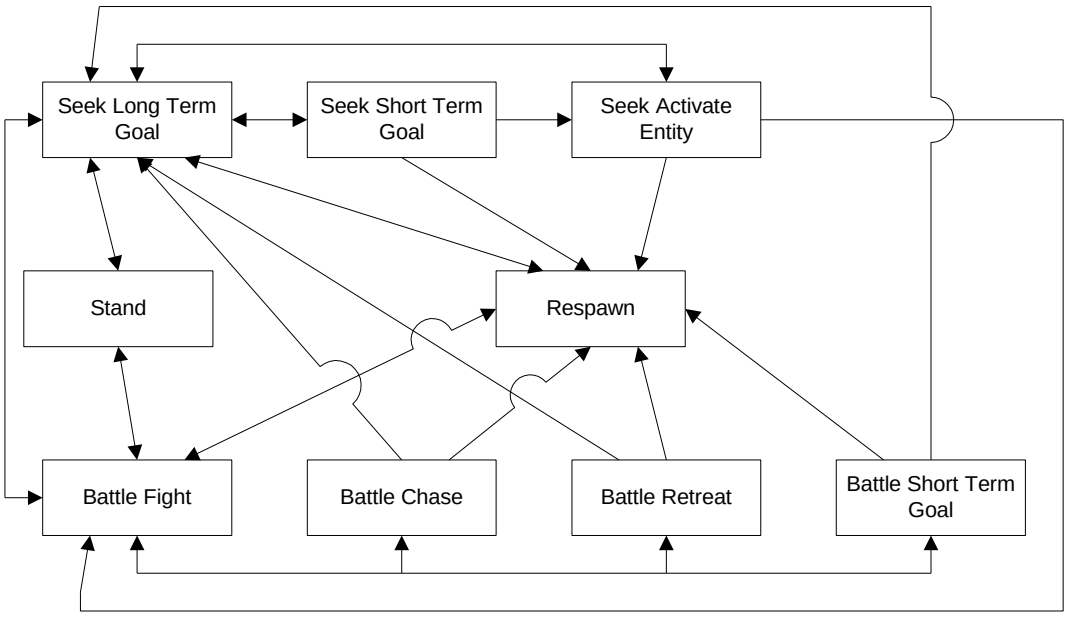


Figure 15.1: AI network.

Figure 15.1 shows the network used by the bot. The square boxes with a name represent the nodes. The conditional links or jumps in the network are represented by the arrows. Some arrows go both back and forth between two or more nodes. However each direction between two nodes along such a link has a different condition for a potential jump. Two nodes are not shown in figure 15.1 because they would only clutter the image. These nodes are for when the bot is in observer mode and when the game is

in the intermission state between two consecutive games. Both in observer mode and when the game is in the intermission state the bot is not playing the game. From every other node the bot can end up in one of these two nodes. When the bot leaves either observer mode or the next game starts the bot continues with the “Stand” node.

15.2 The nodes

The network has the following nodes:

- Respawn
- Stand
- Seek Long Term Goal
- Seek Short Term Goal
- Seek Activate Entity
- Battle Fight
- Battle Chase
- Battle Retreat
- Battle Short Term Goal
- Intermission
- Observer

When the current AI node is any of the nodes except the “Respawn”, “Observer” or “Intermission” node the bot is “alive” and can be killed. When the bot dies in the game the bot will always go to the “Respawn” node. In the “Respawn” node the bot will try to respawn by using the basic action to respawn. As soon as the bot respawns the current AI node will change to “Seek Long Term Goal”.

If the game ends then the bot will go to the “Intermission” node. When the bot ends up in observer mode the bot will go to the “Observer” node. When the game starts or the bot leaves observer mode the bot goes to the “Stand” node.

The “Stand” node is mostly used for when the bot is typing a chat message. While typing a message a player cannot move nor shoot. The bot will go to the “Stand” AI node when the bot chooses to type a message. The actual construction of chat messages and the reaction to messages from other players is described in section 10. When an enemy walks by, when the bot is typing a message, the bot might decide to attack the enemy. In this case the bot will go to the “Battle Fight” node. When the bot is done typing the message the bot will go to the “Seek Long Term Goal” node.

In the “Seek Long Term Goal” node the bot goes for one of the long term goals the bot can have. While going for such a long term goal the bot can pickup nearby items along the way or go for air when swimming. When the bot decides to go for such a short term goal the bot goes to the “Seek Short Term Goal”. As soon as the bot picked up the nearby item or recovered its breath while swimming, the bot will continue with the long term goal and the bot goes back to the “Seek Long Term Goal” node.

While going for a long term or short term goal the bot checks for obstacles and little puzzles that need to be solved in order to retrieve the goal. There is a special node to handle solving puzzles as described in section 14. The bot activates buttons and triggers in the “Seek Activate Entity” node. The goal stack as described in section 14 is also implemented in this node. While trying to solve a little puzzle the bot might go for short

term goals. After retrieving the short term goal in the “Seek Short Term Goal” node the bot will continue with the “Seek Activate Entity” node.

```
/*
==================
AINode_Battle_Fight
==================
*/
int AINode_Battle_Fight(bot_state_t *bs) {
            int areanum;
            vec3_t target;
            aas_entityinfo_t entinfo;
            bot_moveresult_t moveresult;

            // if the bot is in observer mode
            if (BotIsObserver(bs)) {
                        AIEnter_Observer(bs, "battle fight: observer");
                        return qfalse;
            }
            // if in the intermission
            if (BotIntermission(bs)) {
                        AIEnter_Intermission(bs, "battle fight: intermission");
                        return qfalse;
            }
            // respawn if dead
            if (BotIsDead(bs)) {
                        AIEnter_Respawn(bs, "battle fight: bot dead");
                        return qfalse;
            }
            // if there is another better enemy
            if (BotFindEnemy(bs, bs->enemy)) {
            }
            // if no enemy
            if (bs->enemy < 0) {
                        AIEnter_Seek_LTG(bs, "battle fight: no enemy");
                        return qfalse;
            }
            // retrieve informtion on the enemy
            BotEntityInfo(bs->enemy, &entinfo);
            // if the enemy died
            if (bs->enemydeath_time) {
                        if (bs->enemydeath_time < FloatTime() - 1.0) {
                                    bs->enemydeath_time = 0;
                                    return qfalse;
                        }
            }
            else {
                        if (EntityIsDead(&entinfo)) {
                                    bs->enemydeath_time = FloatTime();
                        }
            }
            // if the enemy is invisible and not shooting the bot looses track easily
            if (EntityIsInvisible(&entinfo) && !EntityIsShooting(&entinfo)) {
                        if (random() < 0.2) {
                                    AIEnter_Seek_LTG(bs, "battle fight: invisible");
                                    return qfalse;
                        }
            }
            // position of enemy
            VectorCopy(entinfo.origin, target);
            // update the last seen enemy area and origin if possible
            areanum = BotPointAreaNum(target);
            if (areanum && trap_AAS_AreaReachability(areanum)) {
                        VectorCopy(target, bs->lastenemyorigin);
                        bs->lastenemyareanum = areanum;
            }
            // update the attack inventory values
            BotUpdateBattleInventory(bs, bs->enemy);
            // if the enemy is not visible
            if (!BotEntityVisible(bs->entitynum, bs->eye, bs->viewangles, 360, bs->enemy)) {
                        if (BotWantsToChase(bs)) {
                                    AIEnter_Battle_Chase(bs, "battle fight: enemy out of sight");
                                    return qfalse;
                        }
                        else {
                                    AIEnter_Seek_LTG(bs, "battle fight: enemy out of sight");
                                    return qfalse;
                        }
            }
            // use holdable items
            BotBattleUseItems(bs);
            // travel flags the bot may use for navigation
            bs->tfl = TFL_DEFAULT;
            if (bot_grapple.integer) bs->tfl |= TFL_GRAPPLEHOOK;
            // if in lava or slime the bot should be able to get out
            if (BotInLavaOrSlime(bs)) bs->tfl |= TFL_LAVA|TFL_SLIME;
            //
            if (BotCanAndWantsToRocketJump(bs)) {
                        bs->tfl |= TFL_ROCKETJUMP;
            }
            // choose the best weapon to fight with
            BotChooseWeapon(bs);
            // perform attack movements
            moveresult = BotAttackMove(bs, bs->tfl);
            // if the movement failed
            if (moveresult.failure) {
                        //reset the avoid reach, otherwise bot is stuck in current area
                        trap_BotResetAvoidReach(bs->ms);
                        bs->ltg_time = 0;
            }
            // special AI for when the bot is blocked by an obstacle
            BotAIBlocked(bs, &moveresult, qfalse);
            // aim at the enemy
            BotAimAtEnemy(bs);
            // attack the enemy if possible
            BotCheckAttack(bs);
            // if the bot wants to retreat
            if (!(bs->flags & BFL_FIGHTSUICIDAL)) {
                        if (BotWantsToRetreat(bs)) {
                                    AIEnter_Battle_Retreat(bs, "battle fight: wants to retreat");
                                    return qtrue;
                        }
            }
            return qtrue;
}
```

Figure 15.2: C code for “Battle Fight” node.

When the bot encounters an enemy while the bot is in any of the “Seek” nodes the bot will either fight the enemy or retreat from the enemy. When the bot feels fit enough to

fight the enemy the bot goes to the “Battle Fight” node. In this node the bot will fight the enemy as described in section 13. The C code of the procedure for the “Battle Fight” node is shown in figure 15.2. When the enemy goes out of sight the bot might want to chase the enemy. If the bot feels like hunting down the enemy the bot goes to the “Battle Chase” node. In this node the bot will usually go to the position where the enemy was last seen in the hope the enemy will be visible from there. As soon as the enemy is in sight again the bot will return to the “Battle Fight” node.

When the bot is low on health, or has no decent weapon, or just does not feel fit enough to fight the enemy, the bot might want to retreat. In this case the AI goes to the “Battle Retreat” node. In this node the bot will go for a long term goal that leads away from the enemy and often also a long term goal that will get the bot back in better shape, for instance an health item. When the bot is skilled enough it will also shoot at the enemy while retreating.

While the bot is fighting, chasing or retreating from an enemy the bot can also pick up nearby items and go for air while swimming. When the bot goes for such a short term goal the bot uses to the “Battle Short Term Goal” node. As soon as the short term goal is retrieved the bot goes back to the previous “Battle” node.

The table below shows about 42 seconds out of a bot’s life. The table shows how the bot changes between the different AI nodes based on events and decisions. The current goal the bot pursues in each node is shown as well.

Time (seconds)	Event or decision	Current AI node	Current goal
18.1	The bot named Grunt enters the game.	Stand	-
	Bot spawns.	Stand	-
		Seek LTG	
	Bot decides to retrieve item.	Seek LTG	Retrieve rocket launcher
	Bot decides to retrieve nearby item.	Seek LTG	Retrieve rocket launcher
		Seek NBG	Retrieve bullets
19.9	Picked up bullets.	Seek NBG	Retrieve bullets
		Seek LTG	Retrieve rocket launcher
20.6	Bot decides to retrieve nearby item.	Seek LTG	Retrieve rocket launcher
		Seek NBG	Retrieve shotgun
21.5	Enemy in sight.	Seek NBG	Retrieve shotgun
		Battle NBG	Kill the enemy & retrieve shotgun.
22.7	Picked up shotgun & bot wants to retreat.	Battle NBG	Kill the enemy & retrieve shotgun.
		Battle Retreat	Retreat & retrieve rocket launcher.
23.8	Bot decides to retrieve nearby item.	Battle Retreat	Retreat & retrieve rocket launcher.
		Battle NBG	Retrieve armor shard.
25.5	Picked up armor shard.	Battle NBG	Retrieve armor shard.
	Enemy out of sight & bot decides to chase.	Battle Retreat	Retreat & retrieve rocket launcher.
		Battle Chase	Chase enemy.
25.9	Bot decides to retrieve nearby item.	Battle Chase	Chase enemy.
		Battle NBG	Retrieve armor shard.
28.2	Picked up armor shard.	Battle NBG	Retrieve armor shard.
		Battle Chase	Chase enemy.
31.9	Enemy in sight.	Battle Chase	Chase enemy.
		Battle Fight	Kill the enemy.
32.3	Enemy out of sight.	Battle Fight	Kill the enemy.
		Battle Chase	Chase the enemy.
33.4	Enemy in sight.	Battle Chase	Chase the enemy.
		Battle Fight	Kill the enemy.
33.5	Enemy out of sight.	Battle Fight	Kill the enemy.
		Battle Chase	Chase the enemy.
35.4	Enemy in sight.	Battle Chase	Chase the enemy.
		Battle Fight	Kill the enemy.
36.5	Enemy out of sight.	Battle Fight	Kill the enemy.
	Bot decides to retrieve nearby item.	Battle Chase	Chase the enemy.
		Battle NBG	Retrieve plasma gun.
38.3	Bot picked up the plasma gun.	Battle NBG	Retrieve plasma gun.
		Battle Chase	Chase the enemy.
	Bot decides to retrieve nearby item.	Battle Chase	Chase the enemy.

		Battle NBG	Retrieve shells.
41.1	Bot picked up shells.	Battle NBG	Retrieve shells.
		Battle Chase	Chase the enemy.
43.3	Enemy in sight.	Battle Chase	Chase the enemy.
		Battle Fight	Kill the enemy.
44.7	Enemy out of sight.	Battle Fight	Kill the enemy.
		Battle Chase	Chase the enemy.
	Bot decides to retrieve nearby item.	Battle Chase	Chase the enemy.
		Battle NBG	Retrieve 50 health.
	Bot picked up 50 health.	Battle NBG	Retrieve 50 health.
		Battle Chase	Chase the enemy.
46.0	Enemy in sight.	Battle Chase	Chase the enemy.
		Battle Fight	Kill the enemy.
46.7	Bot decides to retreat.	Battle Fight	Kill the enemy.
		Battle Retreat	Retreat from enemy & retrieve rocket launcher.
46.8	Bot decides to go for 25 health instead of rocket launcher.	Battle Retreat	Retreat from enemy & retrieve 25 health.
47.9	Bot decides to retrieve nearby item. (another 25 health item).	Battle Retreat	Retreat from enemy & retrieve 25 health.
		Battle NBG	Retrieve 25 health.
49.5	Bot picked up 25 health.	Battle NBG	Retrieve 25 health.
		Battle Retreat	Retreat from enemy & retrieve 25 health.
	Bot decides to chase the enemy.	Battle Retreat	Retreat from enemy & retrieve 25 health.
		Battle Chase	Chase the enemy.
51.6	Bot decides to retrieve nearby item.	Battle Chase	Chase the enemy.
		Battle NBG	Retrieve bullets.
	Bot picked up bullets.	Battle NBG	Retrieve bullets.
		Battle Chase	Chase the enemy.
53.5	Enemy in sight.	Battle Fight	Kill the enemy.
53.8	Bot decides to retreat.	Battle Fight	Retreat from enemy & retrieve 25 health.
	Bot decides to go for shotgun instead of 25 health.	Battle Retreat	Retreat from enemy & retrieve shotgun.
55.6	Bot picked up shotgun & decides to go for cells.	Battle Retreat	Retreat from enemy & retrieve shotgun.
		Battle Retreat	Retreat from enemy & retrieve cells.
57.0	Enemy killed by Grunt with the plasma gun.	Battle Retreat	Retreat from enemy & retrieve cells.
		Seek LTG	Retrieve cells.
58.1	Bot picked up cells & decides to go for an armor shard.	Seek LTG	Retrieve cells.
		Seek LTG	Retrieve armor shard.
58.5	Bot picked up armor shard and decides to go for heavy armor.	Seek LTG	Retrieve armor shard.
		Seek LTG	Retrieve heavy armor.
58.6	Bot picked up heavy armor & decides to go for an armor shard.	Seek LTG	Retrieve heavy armor.
	Bot decides to retrieve nearby item.	Seek LTG	Retrieve rocket launcher.
		Seek NBG	Retrieve armor shard.
62.2	Bot picked up armor shard.	Seek NBG	Retrieve armor shard.
		Seek LTG	Retrieve rocket launcher.
62.7	Bot picked up rocket launcher	Seek LTG	Retrieve rocket launcher.
…	…	…	…

The bot starts out by gathering items until it encounters an enemy. The bot first retreats from the enemy because it does not feel fit enough to fight. As soon as the bot has picked up some health, the enemy has gone out of sight, the bot feels better and decides to go in pursuit. From the 31st second till the 36th second the bot has a hard time trying to keep up with the enemy. Every time the enemy comes in sight the bot attacks. However the enemy fires back at the bot at the same time, and the bot gets hurt. In the 46th second the bot is quite hurt and decides to retreat. The bot picks up some items again, and goes back into pursuit in the 51st second. The bot attacks the enemy and gets hurt again. The bot decides once again to retreat but keeps firing at the enemy, and kills the enemy in the process in the 57th second. The bot now continues to gain strength by gathering items.

16. Bot Commands

16.1 Interpreting chat messages

The Quake III Arena bots can be ordered to complete certain tasks in the team based game modes. The bots can also answer specific questions in team play like “where are you?”. Human players but also other bots can type chat messages and tell a bot what to do or ask a question. The bot uses the match templates described in section 10 to make sense out of any incoming chat messages. These match templates give the bot a command or question id when one of the templates can be matched with an incoming chat message. Flags that specify specific properties of a chat message are also retrieved from a template that matches. The bot knows what each match template id stands for and can use it to take actions accordingly. The match template flags are used to retrieve additional information from the chat message.

Below a match template is shown that will match with for instance the chat message: “(MrElusive): defend the red armor”.

```
"(", NETNAME, "): defend ", “the “|””, KEYAREA = (MSG_DEFEND, 0);
```

The name between the braces is the name of the player who typed the message. The message is not specifically addressed to anyone, MrElusive just tells all team mates to defend the red armor. The bot will “understand” the message is about defending a specific area because of the MSG_DEFEND match template id. The name of the player who typed the chat message and the “key area” to be defended are variables in the match template. The bot can retrieve the values of these variables and use them. The bot can retrieve the name of the “key area” and try to find an item with a matching name. The bot can then decide to defend this item. The bot can also retrieve the name of the player who typed the chat message and use this name to respond directly to this player. The bot could for instance respond with “MrElusive I am on my way”.

Below a match template is shown that has a flag to specify a specific property of the chat message.

```
"(", NETNAME, "): ", ADDRESSEE, " defend ", “the “|””, KEYAREA = (MSG_DEFEND, ST_ADDRESSED);
```

This template will for instance match to the chat message “(MrElusive)”: Grunt defend our base”. In this case the match template has a flag ST_ADDRESSED which tells the bot that the message is addressed to someone. When the bot finds such a flag in the template it can retrieve the name of the addressee and compare it to its own name. In this case the name of the addressee is “Grunt”. The bot will probably want to ignore the message if it is not the addressee. Notice that the value of the “key area” variable is now “our base”. There is no “the” in the sentence which is allowed by the match template with the “the “ | ”” construction. The chat message should contain “the “ or an empty string “” at that position.

Since the match templates are stored per context the bot can decide to use only the templates for a specific contexts. The bot can also use match templates from a specific context to try to find a match with the value of a variable retrieved from matching with a

template. For instance if the bot retrieves the value of the addressee variable it can try to find a match with a match template from the following context:

```
MTCONTEXT_ADDRESSEE
{
	"everyone"|"everybody" = (MSG_EVERYONE, 0);
	TEAMMATE, " and "|", "|","|" ,", MORE = (MSG_MULTIPLENAMES, 0);
	TEAMMATE = (MSG_NAME, 0);
}
```

This will allow the bot to figure out if multiple people were addressed with a chat message. The chat message could for instance be “(MrElusive): everybody defend the base”. To parse all the names out of the following chat message: “(MrElusive): Grunt and Hunter defend the railgun”, the bot can use the above context multiple times. First time the value of the addressee variable will match with the MSG_MULTIPLENAMES template. Then the bot can continue with the value of the MORE variable which will match with the MSG_NAME template.

Similarly the time can be parsed from a chat message if a specific time is specified. Below a match template is shown that will match with for instance the chat message: “(MrElusive): Grunt defend the base for 10 minutes”.

```
“(", NETNAME, “): ", ADDRESSEE, " defend ", "the "|"", KEYAREA, " for", TIME =
	(MSG_DEFENDKEYAREA, $evalint(ST_ADDRESSED|ST_TIME));
```

The match template uses $evalint which merges the two (bit) flags together to one value. One of the flags is ST_TIME because a time is specified in the chat message. The bot can now use the following context to make sense out of the specified time.

```
MTCONTEXT_TIME
{
	TIME, " minute"|" min","s"|"" = (MSG_MINUTES, 0);
	TIME, " second"|" sec","s"|"" = (MSG_SECONDS, 0);
	"ever" = (MSG_FOREVER, 0);
	" a long time" = (MSG_FORALONGTIME, 0);
	" a while" = (MSG_FORAWHILE, 0);
}
```

Using this context the bot can understand chat messages like “(MrElusive): Grunt defend the base forever” and “(MrElusive): Grunt defend the base for a while”. In the latter case the bot will give meaning to the expression “for a while”.

16.2 Commands

The Quake III Arena bot “understands” and reacts upon several commands. These commands allow the bots to operate in teams and follow orders from a team leader.

Help someone

Bots can be ordered to help someone. For instance “Grunt help Hunter” or “Grunt help me”. The bot will go to the person in need of help and will try to be of assistance.

Accompany someone

Bots can be ordered to accompany someone. For instance “Grunt follow Hunter” or “Grunt accompany me”. A bot ordered to follow someone will go towards the player who needs company and will follow this player around. When the player to be followed ends up in a fight the bot will assist in the battle.

Defend key area

Bots can also defend a key area. A player or team leader can for instance order a bot as follows: “Grunt defend the base” or “Grunt guard the railgun”. When ordered to do so the bot will defend the key area.

Get the enemy flag

In CTF mode bots can be ordered to capture the enemy flag. For instance “Grunt get the enemy flag”. Grunt will then go of to the enemy base and will try to retrieve the enemy flag.

Return our flag

When the base flag is stolen by the enemy team in CTF mode the bots can be ordered to return the flag. For instance “Grunt return our flag”. Grunt will then go of to find the flag of its team. The bot will try to return the flag to the base by killing the enemy carrying it and picking up the dropped flag.

Rush to the base

In CTF mode bots can be ordered to rush back to the base. When the base is under attack by the enemy a bot can be ordered to come back for assistance. For instance “Grunt rush to the base”. Grunt will then run back to the base trying to eliminate any resistance encountered.

Camp

Bots can be told to camp somewhere. For instance “Grunt camp the railgun”. Grunt will then camp near the railgun. A player or team leader can also order a bot to camp at its current location. For instance “Grunt camp there”. When a bot is supposed to camp at the location of the player who orders, then “Grunt camp here” can be used.

Patrol

Bots can patrol a certain area or a specific number of key areas. A bot can be ordered to patrol by for instance saying “Grunt patrol from the railgun to the rocket launcher to the red armor and back”. Grunt will then walk from the railgun to the rocket launcher to the red armor to the railgun etc. In this case the bot loops in a circle along the three items. If the bot is supposed to go back and forth, a player can order the bot as follows: “Grunt patrol from the railgun to the rocket launcher to the red armor and reverse. The bot will now go from the railgun to the rocket launcher to the red armor to the rocket launcher to the railgun etc. The bot can be ordered to patrol between any number of key areas and/or items.

Get item

Bots can be ordered to retrieve a specific item. For instance “Grunt get the rocket launcher”. Grunt will then go of to get the rocket launcher.

Kill

Bots can be sent after a specific player on the enemy team. For instance: “Grunt kill Hunter”. Grunt will then try to hunt down Hunter.

Lead the way

A bot can be told to lead someone the way towards its goal. For instance “Grunt lead Sarge the way”. If Grunt is not anywhere near Sarge then Grunt will first go to Sarge. From there Grunt will lead Sarge the way, and wait if Sarge cannot keep up. In case Grunt looses sight with Sarge then Grunt will go back to find Sarge.

Dismiss

Bots can be dismissed from their currently ordered task. For instance “Grunt dismissed”. When dismissed, the bot will decide for itself what to do.

Start team leadership

A bot can be ordered to be the team leader. For instance “Grunt you are the leader”. Grunt will then become the team leader and will tell other human and artificial players on its team what to do.

Stop team leadership

The team leader can also to stop being the leader and leave the job for someone else. For instance “I stop being the team leader”. Someone else can also tell that the team leader stops fulfilling that task. For instance “Grunt stops being the leader”. In both cases all the bots on a team will know that the team leader resigned his job and a new team leader can be chosen.

Join sub team

Sub teams or squads can be created within a team. Bots can be ordered to create a sub team or squad. For instance “Grunt and Sarge create team alpha”. Both Grunt and Sarge will then join the sub team named alpha. Bots can also be ordered to join an existing sub team. For instance “Hunter join squad alpha”. All members of a sub team can be ordered to do something by using the sub team name. For instance “alpha defend the base”. All bots in team alpha will now defend the base.

Leave sub team

Bots that are in a sub team can be ordered to leave the sub team. For instance “Grunt leave your team”. Grunt will now no longer be in any team. Bots do not have to leave their current sub team before they can be ordered to join another team.

Task preference

Players can report their task preference to the team leader. For instance “I would like to be on defense” or “I want to be on offense”. A bot team leader will take these preferences into account when assigning tasks to players.

16.3 Questions

The bots can be asked several questions in team based game modes.

Where are you?

A player can ask where a bot is. For instance “Grunt where are you?”. In case the question is directed to a bot on the players team this bot will answer and tell its location on the map. The bot will usually refer to a location near a specific item on the map. For instance “I am near the railgun in the blue base”.

What are you doing?

Bots can also be asked what they are doing. For instance “Grunt what are you doing?”. The bot will answer and tell what its intentions are.

In which team are you?

Players can ask in which (sub) team a bot is. For instance “Grunt in which team are you?”. The bot will answer with the name of the (sub) team it is in. In case the bot is not in a (sub) team the bot will respond accordingly.

Who is the leader?

Players can also ask who the team leader is. For instance “who is the team leader?”. This is useful when a player joins halfway a game while others are already playing. In case one of the bots in the game is the team leader that bot will respond and tell the player that it is the leader. For instance “I am the team leader”.

What am I supposed to do?

A player can ask the team leader what he or she is supposed to do. For instance “what am I supposed to do?”. In case there is a bot team leader this bot will respond and tell the player what to do.

17. Team AI

17.1 Individual team AI

The bot has limited AI to play the team based game modes in Quake III Arena. Using this AI only, a bot does neither care much for, nor actively interact with it's team mates. Of course a bot will never attack nor try to hurt team mates but aside from that the bot does not care much about the team mates when only using this individual team AI.
In regular team play the individual team AI does not provide the bot with knowledge about special team related goals. The bot will gather items and fight enemies as it does in deathmatch. However in the CTF team game the bot does know about special team related goals. Such team goals are: trying to capture the enemy flag, rushing back to the base when the bot carries the enemy flag, accompanying and protecting a team mate who is carrying the enemy flag, defending the base and returning the base flag when stolen. When the bot is not busy with something the bot decides for itself which goal to achieve based on the situation in the game and sometimes randomly when multiple goals are valid. For instance when both flags are at their base the bot can decide to try to capture the enemy flag, or the bot can decide the defend the base. In such a case one of the goals is chosen based on the bots personal preference. Some team goals are only valid in certain situations. For instance the bot can only try to return the base flag when it is stolen by the enemy. In the same line some of the base defense can be dropped when the enemy has stolen the flag.

17.2 Team leader

To make up for the mostly lacking individual team AI there is a team leader. This team leader has a much better overview on the game than the individual bots and human players. This allows the team leader to initiate better interaction between team mates than individual players usually achieve. With a single team leader, obeyed by all players, there is also no need for conflict resolution when individual players have different ideas about the organisation of the team, or goals that certain players should try to achieve.

The team leader can be a human or an artificial player. A bot has a special brain which allows it to be the team leader. This extra brain of the team leader or command center resides at the 4th layer of the structure shown in figure 5.1. The team leader orders the team mates what to do. Aside from being the one to order others what to do the team leader has no special role in the team. The team leader will also assign itself one of the team goals, just like the leader orders team mates to achieve certain goals. To order team mates to do certain things, the team leader uses chat messages that comply with the commands bots understand as described in section 16. This gives the team leader a wide range of commands and orders to organize the team and tell team mates which goals to achieve. The chat messages used as orders are stored in the initial chats as described in section 10.

A bots will always try to obey the orders from a team leader. However in certain situations the bot will ignore orders when pressing tasks have to be completed. For instance when a bot is ordered to accompany a team mate and right after being ordered, picks up the enemy flag. The bot will then ignore the order for the time being and first rush back to the base to score.

In regular teamplay the team leader orders the team members to work in little groups. In these groups one of the players takes the lead and one or more others follow. The size of the groups depends on the total number of players on the team. These little groups go through the level seeking for members of the enemy team in order to take them out.

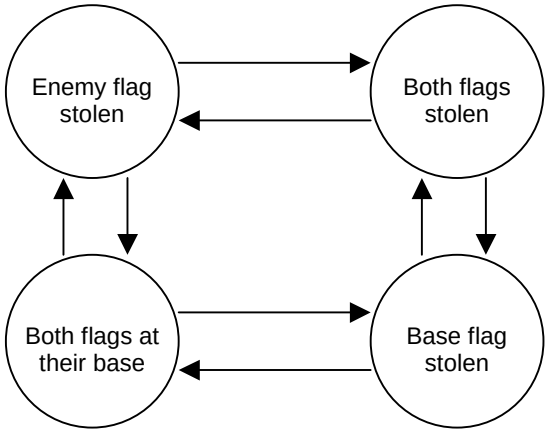


Figure 17.1: Four states of a CTF game.

In the CTF game type the team leader orders team members based on the current situation. Four different situations are distinguished: both flags are at their base, the enemy flag is not at it's base, both flags are not at their bases or the team flag is not at it's base. When both flags are at their base a specific number of team members are ordered to attack the enemy base and try to capture the enemy flag. The remaining team members are ordered to defend the base. When one of the team mates manages to steal the enemy flag the team leader will order some of the other team members to follow the flag carrier. The other team members will go or stay on defense to make sure the enemy gets no chance to steal the base flag. In case the enemy does steal the base flag some of the team members are ordered to return the flag. It is usually also wise to keep some players on defense even though the flag is not at the base. As soon as the flag returns these team members can immediately make sure the enemy does not get a chance to steal the flag again. When the enemy flag is at it's base and the team flag is stolen several team members are sent out to return the team flag and steal the enemy flag. Also here it is often wise to keep some people on defense for when the team flag returns.

Care has to be taken by the team leader that the team mates are not flooded with messages and orders. Players should keep most of their attention to what is happening directly in front of them. They cannot afford to spend a lot of time browsing through a lot of messages from a team leader. Too much information in a message is not good as well. A player should be able to quickly read and understand an order or message. The team leader does not broadcast orders only applicable to specific team members. Each team mate who needs to be ordered individually receives a personal message from the team leader.

18. Implementation & tests

18.1 Implementation

The bot AI as presented in this thesis has been implemented in the C language. Several parts of the game engine run in a virtual machine. Running these parts of the code in a virtual machine allows them to be platform independent. The code that runs in the virtual machines is also publicly available which allows everyone to modify the game. The virtual machine also assures that third party developers cannot do any malicious things from within the code that runs in it.

Part of the bot AI code also runs in the virtual machine. The 3rd and 4th layer of the AI are very dependent on the gameplay rules. To allow third party developers to easily modify the game these layers run in the virtual machine. The 1st and 2nd layer of the AI are far less dependent on the game and game rules. However there is a more important reason not to run the code for those layers in the virtual machine. The virtual machine either interprets the code or does a last minute compile, the latter being faster. Either way running code in the virtual machine has quite an impact on the speed. Running the code for the 1st and 2nd layer of AI in the virtual machine would simply be too slow.

The 1st and 2nd layers of bot AI code is about 33.7 kilo lines of code (KLOC). The 3rd and 4th layer of bot AI code is about 16.1 KLOC. All the bot AI code together accounts for about 25% of all the code in Quake III Arena that is used at run time.

The code for the game engine including the bot AI code has been ported to several different platforms. First of all the game has mostly been developed on x86 compatible PCs. The game has been ported to PPC and also the game platforms PS2 (Play Station 2) and the Dreamcast.

The source code for the 3rd and 4th layer of the bot AI together with the game and client game code is publicly available on the Internet at:
ftp://ftp.idsoftware.com/idstuff/quake3/source/Q3A_TA_GameSource_127.exe

Information about editing the source files of the bot characters included with the game is available on the Internet at:
ftp://ftp.idsoftware.com/idstuff/quake3/tools/q3abotedit.zip

18.2 Bot characters

A total of 32 different bot characters were created for Quake III Arena. There are male and female characters and also biomechanical creatures that all have their own looks and outfit. Each bot character has it's own set of characteristics as described in section 8. The characters also have their own personal preferences for items and weapons in the game. Pictures of the different bot characters can be found in appendix B.

18.3 AAS & Maps

The 3D representation used by the Area Awareness System is pre-calculated. This pre-calculated data is loaded before the game starts, and provides the bot with all the necessary information about routing and navigation instantly. The following table shows for each map in Quake III Arena the characteristics of the pre-calculated data for the Area Awareness System. The data was compiled on an Intel 650MHz Pentium 3 Coppermine.

Map name	# of areas	# of areas used for routing	# of reachabilities	Compile time in seconds
Q3dm1	1714	436	1970	88
Q3dm2	1513	381	2010	82
Q3dm3	1467	519	2128	57
Q3dm4	5208	842	4085	272
Q3dm5	2106	585	2641	86
Q3dm6	2638	708	3147	125
Q3dm7	3606	1107	5263	162
Q3dm8	3263	1016	4818	211
Q3dm9	2633	699	3189	144
Q3dm10	2266	817	3658	110
Q3dm11	6150	1893	7903	502
Q3dm12	5881	1836	7634	342
Q3dm13	2686	734	3161	80
Q3dm14	4343	1456	6732	188
Q3dm15	4273	1189	5794	203
Q3dm16	2081	494	3529	68
Q3dm17	2690	544	2622	56
Q3dm18	2782	889	5106	95
Q3dm19	2057	261	1062	19
Q3tourney1	2680	540	2191	182
Q3tourney2	1787	626	2826	88
Q3tourney3	1711	572	2727	46
Q3tourney4	1924	507	2211	48
Q3ctf1	2183	663	2761	117
Q3ctf2	10317	5230	28181	948
Q3ctf3	3109	870	3978	319
Q3ctf4	3891	720	3240	62

As can be seen in the above table most maps compile within a matter of minutes. That is typically faster than a human player can completely learn how to navigate a specific map. Since creating the data for AAS is quite fast, and routing information can be calculated to virtually any spot on the map, the system was in some cases used during the development of Quake III Arena to find places on a map where players could go, but were not supposed to go. Such places could then be sealed of to avoid players from going there. The above table also shows that the system works with very large maps like “Q3ctf2”. This map has over 5000 areas and over 5 times more reachabilities between the areas. The Quake III Arena bot is able to navigate even these large maps that have a lot of polygonal detail. However creating a BSP tree for these large maps can be somewhat difficult. The algorithm used to create the BSP tree uses 32 bits floating point numbers. These floating point numbers are not exact and come with round-off errors. These errors can be troublesome with very large BSP trees or maps that have a lot of degenerate geometry.

18.4 AAS visualisation

In order to test the Area Awareness System visualisation code is used to visualize the areas. An area underneath an arch is visualized in figure 18.1. In figure 18.2 a route through areas is visualized.

Figure 18.1: area underneath arch

Visualisation code is also used to show the reachabilities between areas. Figure 18.2 shows a ‘jump’ reachability from one area on top of a pillar to another area on top of another pillar. Figure 18.3 shows a ‘jump pad’ reachability.

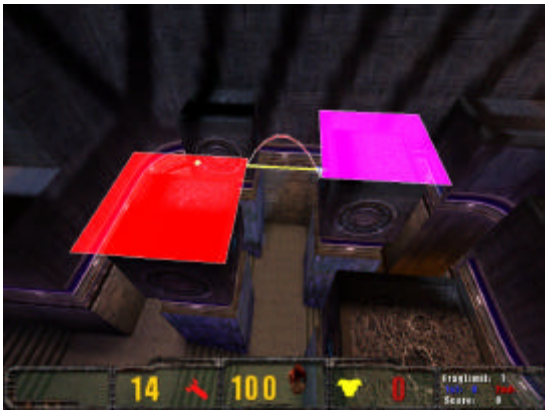


Figure 18.2: jump reachability

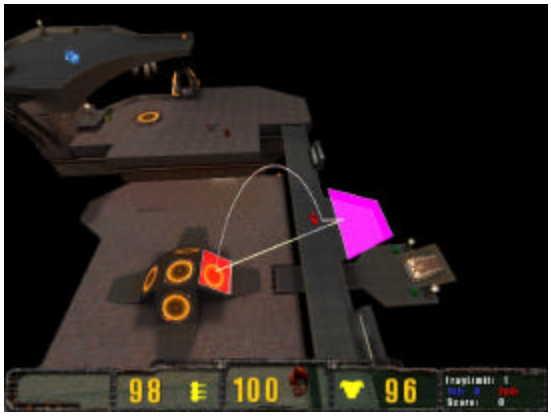


Figure 18.3: jump pad reachability

19. Conclusion

19.1 Bots

The Quake III Arena bot turned out to be a fairly good opponent. The bot is also entertaining and quite suitable for practice and training purposes. The bot is able to navigate through the environment in a life-like manner and the bot can pick up items and handle weapons just like human players. The bot also shows interesting fighting behaviour with tactical moves. The bot can chase opponents that try to escape from a fight, and the bot itself can try to retreat if it is not feeling fit enough to fight.

The bot cannot be distinguished from a human player at first sight. However, with some more effort the bot can usually be identified as being artificial. Human players are often able to recognize certain patterns that are specific to a bot. On the other hand, the evaluation of the bot, based on how hard it is to distinguish the bot from a human player, hardly ever takes place under fair conditions. Human players almost always know in advance if their opponents are human or artificial, based on the name of the player, or simply because of the way the game was started.

The different bot characters can be specified in great detail. The different play styles of the bot characters make them interesting to play with, and their individual chat messages also make them entertaining, as the things they say are usually quite amusing. The bots are especially versatile, because the item and weapon preferences of each individual bot character can be specified, using the fuzzy relations as described in section 9. The numerous characteristic variables described in section 8 also provide a lot of room to create different bot characters.

The bot performs quite well in the team based game modes. The artificial team leader orders other players on the team, and is able to quickly adjusts to changes in the state of a team game. All communication takes place using text messages. The team leader has a wide range of orders available to command team members, and organize the team. The individual bots are able to interpret the text messages, understand the orders and take actions accordingly. The bots in a team can ignore some of the orders if really pressing tasks have to be completed first. A bot remembers an order and the bot will reconsider a temporarily ignored order when the pressing task is completed.

The bot meets most of the gameplay and technical requirements that were set out for. The performance is quite good, and the CPU and memory usage is within acceptable limits. In some rather large maps the routing calculations for the bots can slow down the game simulation somewhat. However these few slowdowns are not significant enough to seriously degrade the gameplay experience.

19.2 AAS

It has been shown that all the information a bot needs in order to navigate and find routes through an arbitrary polygonal environment can be (pre) calculated in a relatively short time, without the need for human intervention. In effect a bot can 'learn' its way around the game world in a very short period of time. This is an advantage over the commonly used waypoint systems. These systems are usually harder to create with programming, and often require human intervention in order to optimize them.

Spatial subdivisions

For the Area Awareness System a BSP tree is used to create a spatial subdivision of the game world. Since deciding whether a polyhedron is tetrahedralizable is NP-complete [15], it seems plausible that finding optimal general spatial subdivisions may be as hard or harder. For AAS a heuristic is used to choose split planes at the nodes of the BSP tree. This heuristic optimizes for the least number of splits and a balanced tree. Using such an heuristic, the most one can practically hope for is good subdivisions that exhibit subquadratic size and near-logarithmic depth for practically occurring environments. Although the spatial subdivisions created for the Quake III Arena maps were not optimal, the total number of convex sub spaces was never too large to handle. The number of areas seems to always stay within practical limits, especially since a lot of convex sub-spaces can be merged, after binary space partitioning is used to create them. Creating a spatial subdivision for huge maps or maps with a lot of degenerate geometry can be troublesome. Due to floating point round off errors the BSP tree created for AAS can sometimes be inaccurate, which might eventually lead to navigation problems for the bot. To avoid these problems the game world could first be subdivided into chunks using a grid. A BSP tree is then created for each chunk, which is significantly smaller and less complex than a BSP tree for the whole game world. The smaller BSP trees created for the separate chunks will likely be less sensitive to floating point round off errors. The BSP trees for all the chunks can then be combined into one BSP tree.

19.3 Future directions

Although the Quake III Arena bot turned out to be a fairly good artificial player there is room for improvements in several areas.

Fighting

The fighting behaviour of the bot could be improved by adding more and better anticipation of enemies. While aiming, the bot could for instance predict which item(s) an opponent is going for. The bot could then shoot missiles at locations along the predicted path of the enemy. Currently, when the enemy is out of sight, the bot always assumes the enemy will come back towards the bot, traveling along the shortest path. However, the enemy does not necessarily have to move back towards the bot. The enemy might for instance want to pick up some item and navigate towards it. Here, the bot could also use better prediction of where the enemy is going. To setup an ambush, the bot would also need to anticipate the movement of the enemy. Where the enemy is going, could be predicted, and an ambush could be setup accordingly. The bot would also want to know from which direction the enemy is likely to approach, in order to face that direction, and have a weapon ready to fire upon sighting the enemy.

To find good locations to ambush an enemy, the bot has to analyze the environment. Certain properties can make a specific location a good position to setup an ambush. The specific properties will have to be identified first before the bot can search for such strategic locations. In the same way the bot could analyze the environment to find other locations, suitable for specific purposes. The bot could search for locations that are best suitable to camp, to defend certain areas, to provide suppressing fire in teamplay or locations that have some other strategic importance.

Planning

In general the bot almost only uses stimulus response behaviour. The bot uses little or no planning to achieve specific goals within the game. Some planning is used when a bot tries to solve a puzzle as described in section 14. Using a goal stack the bot builds up a list of sub-goals, in order to reach the final destination. Most of the time however, the bot only has a single goal. The bot might use sub-goals on the way to achieving its current long term goal, for instance when a nearby item is found that can easily be picked up. However these sub-goals are not planned for, and are only used when encountered. The bot will likely show more intelligent behaviour when it has the ability to construct larger plans. The bot could for instance create a plan to first retrieve specific items before it starts defending a key area.

Team AI

The teamplay behaviour of the bot could be improved upon as well. The bot could be made more aware of certain team goals and tuned for better cooperation with team mates. The chat message parsing could also be improved for the communication with team mates or the team leader. Currently only match templates are used by the bot to 'understand' messages. It is probably worthwhile to use a key based method as used for the Eliza-like reply chats. The reply chats do not only allow the usage of match templates, but also keys to make sure certain words do not occur in the chat message. This might be useful to guarantee certain words do not occur in the variable strings in a match template.
The AI of the team leader could also be improved upon. It is probably useful to create a hierarchy of sub-teams within a team. Each sub-team at a lower level in the hierarchy has it's own lower ranked team leader, much like there are officers, sergeants and corporals in the army. A team leader on a higher level would then order team leaders on a lower level in the hierarchy. Orders will propagate down the hierarchy until individual team members are ordered to complete certain tasks. Such an hierarchy allows a better organisation of a team, and complex tasks to be accomplished more easily. A complex task could be solving a puzzle as described in section 14. A team leader could order other players, or lower ranked team leaders, to activate certain buttons or triggers. Working in a team like this, could significantly reduce the time required to solve the puzzle.

20. References

20.1 Books and articles

1. Larry J. Crocket, *The Turing Test and The Frame Problem*, (1994), Ablex Publishing Corp., Norwood, New Jersey.
   ISBN: 0-89391-926-8

2. C.H. Chen, *The Fuzzy Logic and Neural Network handbook*.
   ISBN: 0-07-011189-8

3. Stuart C. Shapiro, *Encyclopedia of Artificial Intelligence,* (January 1992),
   ISBN: 047150307X

4. Barr and Feigenbaum, *The Handbook of Artificial Intelligence*, vol. 1, (June 1986), Addison-Wesley Pub Co.
   ISBN: 0201168901

5. Nils J. Nilsson, *Principles of Artificial Intelligence,* (June 1986)
   Morgan Kaufmann Publishers
   ISBN: 0934613109

6. Stuart Russel, Peter Norvig, *Artificial Intelligence, a modern approach* (1995) Prentice-Hall
   ISBN: 0131038052

7. L. Boullart, A.Krijgsman and R.A. Vingerhoeds, *Application of artificial intelligence in process control.* (1992) Pergamon Press
   ISBN: 0080420176

8. John David Funge, *AI for Games and Animation*. (1999), A K Peters, Ltd.
   ISBN: 1568811039

9. William van der Sterren, *Terrain Analysis for 3D Action Games*, in Proceedings of the 2001 Game Developer Conference, (2001),
   http://www.cgf-ai.com/docs/gdc2001_paper.pdf

10. John E. Laird, *It knows what you're going to do: Adding anticipation to a QuakeBot*, (1999), University of Michigan.
    http://ai.eecs.umich.edu/people/laird/papers/aticipation-print.pdf

11. Thomas H. Cormen, *Introduction to Algorithms,* (2000 24th printing),
    ISBN: 0-262-53091-0

12. Donald Hearn, M. Pauline Baker, *Computer Graphics*, (2nd edition 1996), Prentice-Hall
ISBN: 013159690X

13. Andrew S. Tanenbaum, *Computer Networks*. (3rd edition 1996), Prentice-Hall
ISBN: 0133942481

14. H. Fuchs, Z. Kedem, B. and Naylor, *Visible Surface Generation by A-Priori Tree Structures*. (July 1980),
Conf. Proc. of SIGGRAPH '80, 14(3), 124-133.

15. Jim Ruppert and Raimund Seidel, *On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra*. (1989), In Proc. 5th Annual ACM symposium on Computational Geometry, pages 380-392.

20.2 Websites

16. Allan Turing Home Page
http://www.turing.org.uk/turing/

17. Fuzzy Logic Laboratorium Linz – Hagenberg
http://www.flll.uni-linz.ac.at

18. Eliza chat program.
http://www.toptown.com/hp/sjlaven/eliza.htm

19. Gamasutra AI index
http://www.gamasutra.com/features/index_ai.htm

20. Binary Space Partitioning page
http://www.cs.buffalo.edu/~whitley/research/graphics/bsp/tutorial.html

21. BSP Faq
http://www.upl.cs.wisc.edu/sigs/gamesig/library/graphics/bsp-faq.html
http://www.dcc.ufba.br/mat056/bsp/bsp.html

22. Dan's programming tutorials BSP
http://members.home.com/droyer/tutorials/Engine03.html

23. BSP Tutorial by Steven Cento
http://www.grandus.com/info/BSPTrees/BinarySpacePartitioningTrees.html

24. Algorithms archive
http://wannabe.guru.org/alg/

25. CTF Communication strategies
http://www.captured.com/articles/comm/

26. Computer Generated Forces by William van der Sterren
http://www.botepidemic.com/aid/cgf/
http://www.cgf-ai.com

27. Bot Epidemic (excellent site with FPS bot related news and info)
http://www.botepidemic.com

28. Artificial Intelligence by Criss Eliasmith
http://artsci.wustl.edu/~philos/MindDict/artificialintelligence.html

29. John Laird's Artificial Intelligence & Computer Games Research
http://ai.eecs.umich.edu/people/laird/gamesresearch.html

30. Soar bot for Quake2
http://ai.eecs.umich.edu/~soarbot/

31. Amit's Game programming information.
http://www-cs-students.stanford.edu/~amitp/gameprog.html

32. PC AI magazine
www.pcai.com/pcai

20.3 Previous work

33. Omicron bot for Quake
http://www.botepidemic.com/gladiator/obots/obots.html

34. Gladiator bot for Quake II
http://www.botepidemic.com/gladiator

Appendix

A. Quake III Arena

A.1 Getting about

Walking
The player can use the 'forward', 'backward', 'move left' and 'move right' keys to move in the specified direction. Turning left or right is done with the 'left' or 'right' keys or by sliding the mouse to the left or right.

Jumping & crouching
The player can tap the 'jump' key to jump. The player jumps further if also moving forward. The player can hold down the ‘crouch’ key to move around in a crouched position.

Swimming
When underwater, the player can “aim” in the direction he/she wishes to go, and move forward. The player has full 3D freedom. The 'jump' key is used to move straight up towards the surface. Once on the surface, the player can tread water by holding down the 'jump' key.

Shooting
The player can tap the 'shoot' key to fire the weapon currently held. The player can hold it down to keep firing.

Using objects
There's no special key to use objects in the environment. The player can push a button or open a door, by walking up to it. To ride a platform, the player can step on top of it. If a door won't open or a platform won't lower, the player might need to do something special to activate it.

Picking up stuff
To pick up items, weapons and powerups, the player has to walk over them. When an item isn't picked up this means the player already has the maximum amount possible of it or the item is inferior to what the player already has.

A.3 Environmental hazards

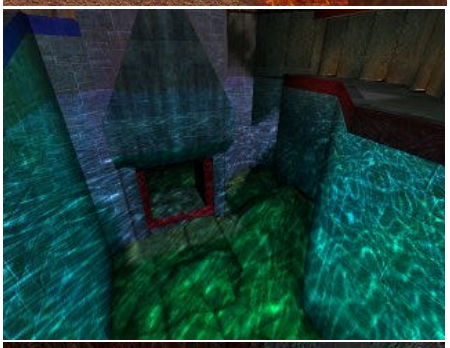
Slime

Hurts the player instantly and keeps on hurting. The player should stay out of slime unless the player has a battle suit.

Lava

The player should keep out of lava because it's very deadly. However the battle suit will also protect the player from lava.

Water



A player can wade through water if it isn't very deep. Deeper waters the player can swim through. The player has to come up for air periodically to avoid drowning.

Death pit

These endless pits are usually filled with fog and the bottom cannot be seen. When falling into a death pit the player is assured of a sudden death.

Shooter

There are several different shooters. In the picture a grenade launcher is placed at a wall. When triggered this shooter throws grenades into the room

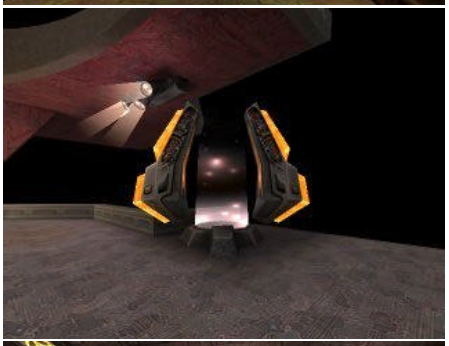
A.4 Structural systems

Door

Most doors open when approached. However some doors need to be opened with a button or some other trigger.

Teleporter

Teleporters instantly transport a player to another location on the map.

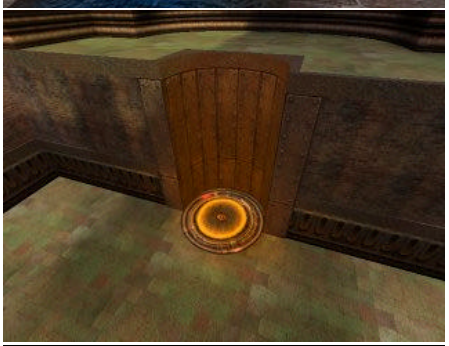


Portal

A different location on the map can be seen through a portal. Often these portal also teleport the player to that location on the map when entered.

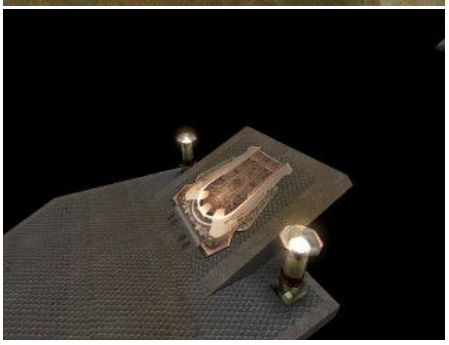
Jump pad

Jump pads push the player up to higher levels in a map.



Acceleration pad

These pads can push the player over long distances in a map.



Bobbing platform

Some maps have bobbing platforms. A player can jump onto these platforms and ride them to higher levels in the map.

Button

Touching them activates buttons. A button is used to open doors and grates, which lead to other areas of the map.

Pendulum

Pendulums swing from side to side. Players die when hit by a pendulum so care has to be taken when trying to pass them.

A.5 Weapons

Gauntlet

The gauntlet is an over sized metallic glove with a spinning blade. The weapon does not require any ammunition. A player will have to get up real close to an enemy to inflict damage. However a successful hit inflicts 50 points of damage.

Machine gun

The machine gun is the most effective default weapon. This instant hit weapon inflicts between 5 and 7 points of damage per bullet. The rate of fire is quite high. The machine gun is pretty accurate but at large distance quite a few bullets will miss their target. The default amount of ammunition for this weapon is 100. The player can pick up and carry around up to 200 bullets of ammunition.

Shotgun

The double-barreled shotgun is a very lethal weapon at close range. Though the weapon appears double-barreled it only uses one shell of ammunition per discharge with a spread of 11 pellets. Each pellet inflicts approximately 10 points of damage. The pellets disperse in a spread pattern, which makes the weapon less effective at a larger range. When a player acquires the weapon it has 10 rounds of ammunition. The player can pick up and carry around up to 200 rounds of ammunition.

Plasma gun

The plasma gun shoots hot blobs of plasma at a pretty high rate. A burst of hot plasma will be quite difficult to avoid for an enemy. Each plasma blob inflicts approximately 20 points of damage. The impact of a blob near an enemy can also inflict splash damage. However the radius of the splash damage is tight. When a player acquires the weapon it has 50 cells of ammunition. The player can pick up and carry around up to 200 cells.

Grenade Launcher

The grenade launcher fires grenades that bounce around for about three seconds before they explode. The grenade explodes immediately upon striking a player. Littering the floor with grenades on a retreat can be very helpful when trying to shake an enemy of your tail. Each grenade can inflict up to 100 points of damage. The grenades also inflict splash damage when the enemy is within range. When a player acquires the grenade launcher it has 10 rounds of ammunition. The player can pick up and carry around up to 200 grenades.

Rocket Launcher

The rocket launcher is one of the most lethal weapons in the game. It fires rockets that inflict around 100 points of damage on impact. When the rocket explodes near a player it will still inflict splash damage. When a player acquires the rocket launcher it has 10 rounds of ammunition. The player can pick up and carry around up to 200 rockets.

Lighting gun

The lightning gun shoots a beam of lightning that inflicts about 80 points of damage for each second a player is hit by the beam. Enemies are ensured of a certain death when the beam is held onto them for several seconds. The beam is limited in range so enemies that are far away can get away unharmed. When a player acquires the lightning gun it has 60 rounds of ammunition. The player can pick up and carry around up to 200.

Railgun

The railgun is the most powerful instant hit weapon. A slug that inflicts 100 points of damage on a direct hit is ejected at an extremely high speed. However the rate of fire is rather low. The railgun is very accurate even over large distances. When a player acquires the railgun it has 10 rounds of ammunition. The player can pick up and carry around up to 200 slugs.

BFG10K

The BFG10K fires green blobs at high speed that inflict 100 points of damage on impact. The impact of the blob also causes splash damage to nearby enemies. The rate of fire is relatively high. Higher than for instance with the rocket launcher. The BFG10K is only found in very few maps and is often hard to acquire. In return the player gets one of the most powerful weapons in the game. When a player acquires the BFG10K it has 20 rounds of ammunition. The player can pick up and carry around up to 200 rounds of ammunition.

A.6 Items & Powerups

Items and power-ups that can be picked up are spread throughout the environments in the game.

Items

5 health
The player can pick up as many of these health items as can be found. Each green health will add 5 to the player's health.

10 health
The yellow health increases the player's health with 10. This item cannot be picked up when the player has 100 or more health.

25 health
This item adds 25 to the player's heath. Also this item can only be picked up when the player's health is below 100.

Armor shard
When picked up the armor shard increases the player's armor by 5. Armor can be picked up while the player has less than 200. All armor above 100 slowly drains away.

Yellow armor
The yellow armor adds 50 to the player's total armor.

Red armor
Picking up a red armor adds 100 to the player's total armor.

Powerups

Mega health
This powerup gives the player 100 points of additional health. After a few seconds all health the player has above 100 will slowly drain away.

Battle Suit
Wearing this powerup the player is only affected by direct projectile hits. All direct hits will also only apply half the damage. The player can also stay underwater while using this powerup

Quad damage
A player wearing this powerup delivers four times the normal damage with weapons.

Haste
When picked up the player can run and shoot 1.3 times faster for half a minute.

Flight
With this powerup the player can fly around the level for a minute.

Invisibility
When picked up renders the player almost invisible.

Holdable items

Holdable items are not used immediately when picked up. The player has to press a special use key to use the item. A player can only carry one holdable item at any time.

Med kit
Using this powerup will fill your health back up to a hundred. The powerup cannot be used when you have 100 or more health.

Personal Teleporter
When this powerup is used the player teleports to a random spawn location in a map. This powerup is ideal to teleport out of a battle if you aren't dong all that well.

A.7 Deathmatch

In deathmatch mode the goal is to kill the other players in the game. Everyone is on his/her own. This game mode is also referred to as Free For All (FFA). The score of a player is increased for every kill. These kills are often called frags. When a player suicides the frag count for that player is decreased. It is possible to set a time and frag limit. When one of these limits is reached the game stops. The winner is the one with the highest frag count.

A.8 Teamplay

In teamplay mode almost everything is the same as in deathmatch mode except there are two teams with players. There is a red and a blue team. A player joins either the red or blue team when entering the game. A player tries to kill all members of the other team. The winning team is the one with the highest accumulated frag count of all players on that team. In teamplay mode it is also possible for a player to send messages to only the members of the player's own team.

A.9 Capture the Flag

In the Capture The Flag game mode (CTF) there is also a red and a blue team with players. There is a blue and a red flag in the game, one positioned in a blue and one in a red base respectively. The red team tries to infiltrate the blue base, capture the blue flag and bring it back to their red flag in their base. The blue team tries to capture the red flag and bring it back to their blue flag in their base. A team scores a point when they capture the enemy flag and bring it back to their base. However their own flag also has to be there. A team cannot score when their own flag is not at its base. While trying to capture the enemy flag and trying to prevent the enemy from capturing their own flag, players can fight with each other as in a regular teamplay game. When a player carrying a flag is killed the flag is dropped at the position where the player died. Another player can pick up the flag from there. If the flag of the player’s team is picked up it returns immediately to the base and the player will not carry the flag around. When a flag is dropped and not picked up within a certain amount of time the flag will also automatically return to its base.

B. Bots


Anarki


Angel


Biker


Bitterman

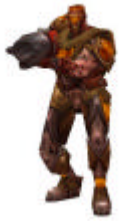

Bones


Cadavre


Crash


Daemia


Doom


Gorre


Grunt


Hossman


Hunter


Keel


Klesk


Lucy


Major


Mynx


Orbb


Phobos

Patriot

Ranger

Razor

Sarge

Slash

Sorlag

Stripe

Tankjr

Uriel

Visor

Wrack

Xaero

C. Terms and abbreviations

3D – Three dimensional.
AAS – Area Awareness System, system that provides the bot with game world state info.
AI – Artificial Intelligence.
bot – Abbreviation for robot.
brush – Convex building block used to build maps in Quake III Arena.
BSP – Binary Space Partitioning.
CPU – Central Processing Unit.
CSG – Constructive Solid Geometry.
CTF – Capture The Flag, one of the teamplay game types in Quake III Arena.
FFA – Free For All, deathmatch game type.
FPS – First Person Shoot-em up.
FSM – Finite State Machine.
GA – Genetic Algorithm
NN – Neural Network