UiO : **Department of Informatics**
University of Oslo

Load Generation for Investigating Game System Scalability

from Demands to Tools

Stig Magnus Halvorsen

Master's Thesis Autumn 2014

Load Generation for Investigating Game System Scalability

Stig Magnus Halvorsen

June 26, 2014

“This book is dedicated to anyone and everyone who understands that hacking and learning is a way to live your life, not a day job or a semi-ordered list of instructions found in a thick book.”

— Chris Anley, John Heasman, Felix “FX” Linder, and Gerardo Richarte [2, p. v]

Abstract

Video games have proven to be an interesting platform for computer scientists, as many games demand the latest technology, fast response times and effective utilization of hardware. Video games have been used both as a topic of and a tool for computer science (CS). Finding the right games to perform experiments on is however difficult. An important reason is the lack of suitable games for research. Open source games are attractive candidates as their availability and openness is crucial to provide reproducible research. Because researchers lack access to source code of commercial games, some create their own smaller prototype games to test their ideas without performing tests in large-scale productions. This decreases the practical applicability of their conclusion.

The first major contribution of the thesis is a comparative study of available open source games. A survey shows a list of demands that can be used to evaluate if a game is applicable for academic use. The study unveils that no open source game projects of commercial quality are available. Still, some open source games seem useful for research. These can be used for the implementation, testing, and verification of scientific concepts. The comparative study suggests that *Doom 3*, *OpenArena* (*Quake III Arena*), and *PlaneShift* are some of the best candidates available as of today.

None of the suggested games are ideal for testing server or network specific concepts, as none provides tools for simulating user-generated load. This is needed to perform a load test on a server, to evaluate the effect of the implemented concepts. Network traffic and system load can be generated through the implementation of clients controlled by artificial intelligence (AI) that simulates real players. They should produce real network traffic and server load that is similar to traffic produced by a real human player. These specialized clients are often referred to as *virtual clients*.

The second major contribution of this thesis describes the process of converting the open source video game *Quake III Arena*, into a user-friendly load generation tool for investigating game system scalability. This has been done through the implementation of *virtual clients*, providing researches with an automated procedure to evaluate network and server performance under various loads. The tool proves to both generate authentic traffic and server load.

Contents

List of Figures

List of Tables

Preface

I will like to thank my supervisors, Kjetil Raaen & Carsten "Griff" Griwodz, for their supervision. This work could not have been done without their invaluable will for sharing their knowledge, their strict demands for academic perfection, and their interest in my work and me. It has been a true honor to work with you both!

Special thanks goes to all my colleagues at the NITH, soon to be the technology department of *Westerdals Oslo ACT*. You considered me a talent and I was requited for your Master's candidate internship. You provided me the currently best job I have had with my own desk, equipment, tons of coffee, all I need for my degree, and friends for life. My colleagues are some of the most talented and friendly people I have ever met, who have together with my position probably taught me more than all the classes I had during the degree. I wish you all the best, and urge you to continue with your Master's candidate internships.

Thanks to *Simula Research Laboratory* for allowing me to steal the time of their legendary *Griff*, and for providing me with my own desk, free lunch, and even more tons of coffee. The researchers at Simula are some of the most knowledgeable computer scientists I have ever met, and I am grateful for the ones who involved with my degree.

I am also grateful for the interesting classes and opportunities at the *University of Oslo*. Special thanks goes to my co-students that improved my learning outcome and kept me motivated; Khiem-Kim Ho Xuan, Georgios Patounas, Eva Andritsopoulou, Thiseas Mengos, Jan Anders Bremer, Kristian Skarseth, Peder Thorup, Erik Thune Lund, Endri Hysenaj and Kristian Hiorth.

Thanks to my family and friends who is always there for me. To my mother, Sølvi Halvorsen, you taught me how to prioritize and focus. To my father, Sverre M. Halvorsen, you taught me to never give up when facing demanding challenges. To my sister, Silje E. Arlund, and her family, who reminds me that life is more than work. To my girlfriend, Maria N. Baugstø, you remind me to stay human. To Martin Nordal, for all the laughs and the introduction to computer science. I would also like to thank *Oslo Sipalki Klubb* and *my* dog, Nemi, for spicing up life.

I would also like to thank the *OpenArena* community who helped me understand the *ioquaake3* engine and *OpenArena*'s project structure. For those that have not been mentioned but that are a part of my life, I am truly grateful for your contributions as well. I would never have made it this far without you. We are all individuals, but it is the ones next to you that mold you.

Part I

Introduction

Chapter 1

Introduction

This thesis describes the modification of an existing open source game into a server and network load generation tool, which can be utilized by scientists to test their concepts in a large-scale system. This chapter explains the motivation behind the thesis, presents the project statement, and provides an overview of the thesis' various topics.

1.1 Motivation

Decades have passed by since the first commercially produced and published video game was released, giving birth to the video game industry. Since then, both the popularity and demand for video games has exploded with an insatiable audience demanding more content and more technically advanced features for every major release. This has resulted in one of the world's toughest industries with high development costs and high risks for failure. The industry is thus a technology promoter that keeps pushing hardware and software to the limit in order to satisfy the ever-demanding market. It is the demanding nature of video games that makes them interesting for scientists, both as tool for research (such as Petlund et al. [48], Raaen et al. [50]), as well a topic of research (such as Claypool & Claypool [12], Eisert [46], Waveren [73]).

In Halvorsen and Raaen (2014)[29] we describe how computer scientists with interest in video games need a freely available and open source video game of similar quality to a modern, commercial game. Unfortunately, there are currently no such open source games available. However, some older open source games exist that may fulfill the needs of many researchers, and these will often be better alternatives than small project specific prototypes. Identifying these games is however difficult as it requires a set of defined requirements and qualities to evaluate if a game is applicable for academic research.

Finding a potential game for research is usually not enough. Research such as Raaen et al (2012)[50] and Petlund et al (2008)[48] requires support for *virtual clients*; AI controlled clients that produce network traffic in the same manner as a real player. This is needed to automatically generate network and server load, enabling the researchers to test their implementations without real players. An open source game with implemented support for virtual clients is thus an ideal tool for such research.

1.2 Project Statement

This thesis aims at identifying modern open source games that are representative of commercial games. Open source games will be evaluated against a specific list of features and qualities that are considered common requirements by computer scientists. The list is developed by evaluating data from a quantitative survey aimed at computer scientists.

Open source games shall be evaluated against the feature list to identify an ideal candidate for implementing support for virtual clients. This converts the selected game into a user-friendly load-generating tool.

1.3 Outline

The thesis is divided into the following chapters, each briefly described below.

You are currently reading the first chapter, which introduces the thesis. The chapter explains the motivation, presents the project statement, and provides an overview of the various topics discussed within the thesis.

The second chapter is located in section 2 on page 9, and provides thorough background information on the concept of video games and how they work. Game engines are presented, and guidelines are given for developing and maintaining high quality game engines.

Chapter three in section 3 on page 19 starts by giving a brief history of video games and its earlier use in computer science. It moves on to discuss the usage of video games in modern computer science, and provides research-based examples for the most common topics. The chapter concludes by evaluating how the existing research would benefit from an open source load generation tool, which narrows down to a thorough explanation of the project statement.

The fourth chapter aims at identifying suitable games for computer science, and is found in section 4 on page 37. An online survey was distributed among computer scientists and evaluated in order to produce a list of requirements and qualities to identify games applicable for research. Examples on how to use this list is provided and three games are identified as potential games for academic work. *Quake III Arena* is presented as an acceptable candidate for the virtual client implementation, and a brief technical overview of its engine is provided.

All technical details of the actual virtual client implementation in *Quake III Arena* are covered in chapter five, section 5 on page 49. It explains how a proxy-like solution solved problems of high complexity, and provides code samples with explanations for the most critical parts of the implementation.

Chapter six in section 6 on page 61 explains how to install and use the virtual client feature of *Quake III Arena* as a tool for network and server load generation. It presents technical documentation on how to retrieve and build the source code, and run the compiled game. Several provided shell script files are also presented and examples illustrate how to use them for load generation.

The seventh chapter presents and discusses the measurements of gathered data from the virtual client implementation in *Quake III Arena*, located in section 7 on page 69. It evaluates both the developed load generation tool and the identification procedure of games for research. Both positive and negative aspects are considered in the evaluation.

Chapter eight in section 8 on page 79 concludes the entire thesis by summarizing its contributions and evaluation results. Opportunities for future works are presented, and may be used in further research on games and load generation tools for research.

Part II

Background

Chapter 2

Background

This chapter aims at providing a detailed explanation on the need for virtual clients in an open source game such as *Quake III Arena*. It provides an in-depth description of video games, explains their use in computer science, and gives an overview of modern challenges of video game research. The content is built upon existing research and literature to ensure correct definitions and credible arguments in its discussions.

2.1 Defining Video Games

> **game**^{1} /geim/ *n* **1** [C] (**a**) form of play or sport with rules: *popular children's games* ○ *a game of chance/skill*. (**b**) instance of this: *to play a game of chess, football, hide-and-seek, etc* ○ *Let's have a game of snooker*.
>
> — *A S Hornby [31, p. 507]*

Oxford Advanced Learner's Dictionary of Current English (1989)[31] defines the term *game* as a form of rule-defined sport or play. More specific examples are provided, where most describe a rule-defined activity or process with human participants. You will probably find similar definitions in other dictionaries and most literature, but very few mention the requirement of fun [36].

> Looking up "game" in the dictionary isn't that helpful. Once you leave out the definitions of referring to hunting, they wander all over the place. Pastimes or amusements are lumped in with contests. Interestingly, none of the definitions tend to assume that fun is a requirement: amusement or entertainment at best is required.
>
> — *Raph Koster [36, p. 12]*

Raph Koster developed a definition of games in his well-recognized book entitled *A Theory of Fun for Game Design* (2005)[36]. He states that games are perceived as fun because they educate the player through practical challenges, which is an approach that applies to all humans. Most games can be considered interactive experiences consisting of sequence patterns with continuously increasing difficulty. It means that games provide a continuous series of challenges that the player learns to master. The player learns the patterns, and is hence educated by the game. Games can therefore be considered interactive educators [36].

> Games are puzzles to solve, just like everything else we encounter in life. They are on the same order as learning to drive a car, or picking up the mandolin, or learning your multiplication tables. We learn the underlying patterns, grok them fully, and file them away so that they can be rerun as needed. The only real difference between games and reality is that the stakes are lower with games.
>
> — *Raph Koster [36, p. 34]*

Games could hence be defined as fun interactive rule-defined educational experiences that involve human actors (players). This is also true for most *video games*, adding electronic processing as a requirement. Finding a generalizable definition of the term is not much easier, but one may consider looking it up in a dictionary. The old Oxford dictionary does not list video games as a superset of games but provides a brief separate entry for *video game* that describes a them as PC powered games generating player controlled images on a TV [31]. This is a quite restrictive definition, as it demands the use of a PC, TV, and images as output. A better description can be found in Jason Gregory's book entitled *Game Engine Architecture*:

> In academia we sometimes speak of "game theory," in which multiple agents select strategies and tactics in order to maximize their gains within the framework of a well-defined set of game rules. When used in the context of console or computer-based entertainment, the word "game" usually conjures images of a three-dimensional virtual world featuring a humanoid, animal, or vehicle as the main character under control.
>
> — *Jason Gregory (2009)[25, p. 8]*

J. Gregory also states that most computer scientists will not use the *video game* term, but rather *soft real-time interactive agent-based computer simulations* [25]. This terminology is discussed further in the next section. However, looking at modern research reveals that the terms *computer game* [46], *online games* [48], and even just *game* [44, 50] have become accepted terms in the computer scientific academia.

For simplicity, all uses of the terms *video game*, *computer game*, *game*, and the more complex previously discussed term are in thesis considered a reference to interactive software, with two- or three-dimensional graphical environment as primary output. Also, this work concerns software, and assumes that the necessary hardware, drivers, and external resources such as network all are in place and properly working for the various features to be fully functional. Video game internals and components are therefore considered terms describing the various software systems that define a video game.

2.2 Video Game Internals

The previous section discussed the formal definition and the generic description of a video game. This section aims at describing the more technical aspects. Video games are as mentioned, interactive software, but what separates them from other software? Gregory's [25] academic definition provides an answer, and a detailed explanation can be found by reverse-dissecting it:

> Most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*.
>
> — *Jason Gregory [25, p. 9]*

Every video game that presents the player with a two- or three-dimensional graphical display is considered a *simulation*, as it attempts to simulate either a real or fictional scenario. The simulation can be as simple as a text based cave (e.g.: *Colossal Cave Adventure*, *Dungeon Crawl*, or *Rouge*), a 2D simulation of a card game (e.g.: *Windows Solitaire*), or as complex as a close to real-life simulation of a city (e.g.: *Battlefield 4* or *Unreal Engine 4*). Pure audio-only games may also be simulations as they can use audio-based techniques to simulate an environment. In general, a video game needs an internal mechanism to perform simulations, whether it's three-dimensional sound, three-dimensional graphics, or another creative media of expression. This work considers mainly visual output. Video games use calculations to generate and project a two-dimensional image consisting of pixel data onto the monitor. This process is known as *rendering*, and modern video games utilize third-party code libraries for rendering. Two popular libraries are *OpenGL* and *DirectX* [69] that utilize dedicated graphics processing unit (GPU) to concurrently perform the heavy graphical computations.

Video games are naturally *computer* based, as they are made to run on digital computer architectures. It may be your custom PC, a dedicated game console (*PlayStation*, *XBox*, etc.), a server in a cloud-cluster, or other forms of dedicated gaming hardware. These games are in other words completely dependent of digital processing. Some video games are available in executable form for multiple operating systems (OS) and CPU architectures.

Agent-based refer to the academic term *agent*, which is used to describe an individual participant in *artificial intelligence* (AI). A game needs participants, or *players*, to be a game. Every participant, including both human controlled and computer AI controlled entities are considered agents. An agent is thus an intelligent in-game entity that exists within the simulated world.

Games are *interactive* software allowing the agent to dynamically alter the software's state (game state) by implementing mechanisms for interaction. This includes capturing user input (e.g.: player keyboard input), and the agents' abilities to interact with each other and the world they reside in. An AI controlled agent may decide to trigger an in-game button that opens a door, hence affecting the game's state.

A requirement of interactivity is *real-time* processing. An in-game interaction occurs, the software processes the interaction and outputs an audio-visual result. In terms of a game, the result is often considered the final simulation output. You press the fire button and the result is a shooting animation, audio playback, and the creation of a new visual entity; a projectile. All needs to occur close to real-time. The alternative is pre-calculated simulations, where interaction is more or less impossible. It would in other words be more like a video, where the output is the same every time you run it.

The real-time requirement introduces deadlines in terms of human perception. Most games and interactive software attempts to be perceived as running in real-time while they are actually not. Every update in a game requires some time-consuming processing, which is usually too fast for humans to notice. The real-time illusion is reduced or destroyed when humans notice the processing delay, indicating that the real-time deadlines are not met. Missed deadlines are usually caused by hardware or network. Complex games running on hardware with insufficient processing capability will cause missed deadlines. Multiple missed deadlines will reduce the real-time experience because output is generated at a too low rate.

High network *latency* is another cause of missed deadlines. Latency is the delay between an action and a response, and is often in the context of online games considered the time it takes for an action to be transmitted, processed, and acknowledged by the recipient. Game sessions with high latency will lead to missed deadlines in terms of lost or delayed network packets. This makes the game unable to project remotely controlled actions within the real-time requirement, potentially leading to unexpected game-dependent artifacts that reduce the player's experience.

The two types of missed deadlines are completely different, but both produce situations that players often refer as *lag*. Lagging gameplay does not necessarily ruin the experience because they may be of little significance, compensated by the engine, or only affect few clients of an online game session. This implies that video games have *soft* deadlines, as there are no catastrophic results if a deadline is not met [25].

In summary from an academic perspective: Video games are computer processed interactive simulation software with soft deadlines, which are dynamic through the integrated support for intelligent agent interactions.

2.2.1 The Game Loop & Frames

There is unfortunately very little scientific literature available covering the core mechanics and theory behind video games. Fortunately, Gregory's academic definition implicitly reveals the core technical requirements. A video game will need to process agent interactions according to the game's rules, which update the game's state and requires it to re-generate its resulting simulation. This needs to be done over and over in order to give a realistic real-time experience, and it continues until the software is terminated. This loop has come to be known as the *game loop*, and every single game implements it in one way or another.

A single iteration of the game loop, from the first instruction to a complete rendered image, is known as a game *frame*. A game loop's performance can be measured in the unit *frames per second* (FPS), where the higher is better. It is a term used to describe the performance of a game and can be used as a benchmarking unit. Good games should have aim at a frame-rate of 50-60 FPS, giving one iteration 1/60 of a second to complete. Even the simplest user actions require a couple of frames, so a high frame-rate is needed to preserve the real-time illusion with high responsiveness and smooth rendering. Hovering an in-game button will for instance need to render multiple cursor transitions and a final hovering effect. Not all games are fast enough for 60 FPS, so they often adjust their frame-rate to 30 FPS as a solution [1].

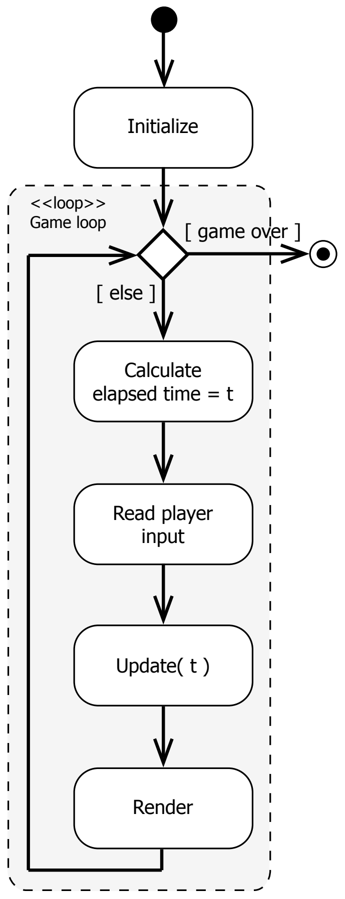


Figure 2.1: UML activity diagram of an uncoupled single-threaded game loop

Valente, Conci, and Feijó's (2005)[72] research on real-time game loops for single player games states that there is difficult to find good literature on the topic. Most commercial developers do not reveal their implementations, and there is very little academic work available. They provide an evaluation of known game loop models, and present a framework that accounts for real-time requirement issues. The models will not be presented in detail, but they all consist of the same tasks defining a game loop; read player input, update the game's state and entities, update the output (render a new frame). The better models also include some sort of synchronization step to make the game run at the same frame-rate on different hardware [72]. Figure 2.1 illustrates a simple game loop, which is an UML adaption of Valente et al's uncoupled single-threaded model[72].

One of the heaviest tasks of a modern game is the graphics processing.

Most, if not all, modern computers are built with dedicated hardware made for processing graphics. They are known as *graphical processing units* (GPU), and are meant as a parallel co-processor to the central processing unit (CPU). It allows the CPU to perform heavy game logic computations while the GPU processes graphics. Graphics processing and generation of visual output in games is known as *rendering*, and the specialized code responsible for the output is often referred to as the *renderer*.

The real-time frame-rate requirement makes video games demanding pieces of software, especially the newest state-of-the-art games that attempt to fully utilize all available modern hardware. Games may generate heavy load on a system where simple modifications can yield significant performance differences. They are hence good testing platforms where researchers can measure the effects of algorithmic changes and conceptual implementations in both software and hardware.

2.2.2 Game Engines

The term *Game engine* is widely used, yet lacks any official or generally agreed definition [25, 69]. It may be because it should be considered an idea rather than a well-defined term [69], even though the game industry [62] and academia [21, 44, 50] both have embraced it. The term is usually used to describe a software system for video game development that handles most, if not all, of the complex sub-systems behind a functional game. A complete engine can be considered a large software framework providing an abstraction layer of complex or system dependent tasks, which allow developers to focus entirely on gameplay and content development. This section aims at describing the background of game engines, why they are useful, and to identify common features of high quality engines.

The term originates from the middle of the 1990s, and came to life together with the release of *id Software*'s first-person shooter game; *Doom* [25]. It was a reference to the game's well-designed software architecture, a highly modular implementation that made a clear separation of the software's different components. *id Software* used this to their advantage and made the gameplay components of the software publicly available (open source), allowing anyone to create custom modifications (*mods*) of their game without revealing the engine's inner mechanics. It became the birth of the modding community [25]. *id Software* continued this trend for all their following releases, and they have publicly released the entire codebase of most of their major games.

Doom's architecture clearly separates the game's individual components and its core mechanics (kernel). It allows developers to implement gameplay specific functionality (game rules) as an independent component rather than a hard-coded part of the kernel. This makes the engine significantly more dynamic, allowing one to modify the game or create novel gameplay without the need to access the core functionalities. The engine has hence a data-driven architecture where the gameplay component utilizes the game's kernel mechanics to independently control the software's output [25].

The kernel is the foundation that binds all the components together and provides them with a cross-component communication channel. It is often the kernel that is referred to when talking about a specific game's engine. The kernel alone does not make a game, but works more like a framework or an *application programming interface* (API) for its components. Gameplay is typically implemented in a separate component, which utilizes the kernel to create functionality that defines the game's rules. Most games also require dynamic game content (3D models, audio files, scripts, etc.) to combine and produce the final audio-visual output. These are often binary files or text files that are difficult to interpret with the naked eye. This implies the need for specialized game content development tools, which often are provided with an engine. Therefore, a complete game engine often refers to a larger software suite consisting of the kernel, engine compliant game development tools, and examples of gameplay components that utilize the kernel and game content. Content development tools, such as level editors, scripting tools, etc., are used to produce game content. The tools usually provide user-friendly interfaces for creating and editing game content, enabling people with limited technical capabilities to work on content creation [69].

There are however no rules defining what separates a game's engine from its content, this combined with the vague term allows potentially even the simplest video games to be considered a game engine. Both Gregory [25] and Thorn [69] propose the same solution by adding the requirement of re-usability. A hard-coded game that typically consists of special-case code is not a game engine as it is difficult, or even impossible to re-use for the creation of new games. True game engines are on the other hand highly re-usable software development kits which allows the developers create almost any game possible and to focus on content production [25]. Thorn presents a simple test to evaluate game software as an engine or not:

> Presented with any library or software, such as DirectX or OpenGL, the developer can proceed to ask, "Can I make a game with this and *only* this?"; if the answer to that question is in the affirmative, then it either *is* or *can be used as* a game engine. It is not enough to show that only in combination with other tools could it make games. The software as it is must be capable of producing games on its own merits to be considered a game engine.
>
> — *Alan Thorn [69, pp. 16-17]*

Thorn also identifies the qualities and expected features of the common well-designed game engines. A good engine separates its functionality into different components, all responsible for a sub-system of the engine. Thorn suggests that the following components are common requirements for most games and should thus be present in a re-usable engine:

- Resource manager (file system)

- Render manager (graphics out)
- Input manager (user input)
- Audio manager (audio playback)
- Error manager (crash catcher)
- Scene manager (coordinate system)
- Physics manager
- Script manager

Each component is responsible for one specific task and nothing more. This makes the entire engine a layered architecture, where the engine is at top delegating tasks through the managers, which again might utilize third party libraries [69]. The architecture provides opportunities for good software design through high cohesion and low coupling.

The kernel defines the remaining functionality of the engine. It is required to initialize the various components, run the *game loop*, and to provide an interface for the game content, binding the various engine components together. Components should know as little about each other as possible, following the design principle of loose coupling. This enables the components to be modular, ideally allowing a component to be modified, added, replaced, or even removed without affecting other components or the kernel. Modularity can be achieved through the implementation of an abstract communication model between the components. The abstract approach enables the components to indirectly utilize each-other. An example is to provide the various components with an abstract interface for performing render calls, allowing the developers to replace the entire render manager without touching any other part of the engine. Adding an indirect communication route through the kernel can ensure this, where the kernel maps component specific functionality to specific kernel calls. A component may then perform a kernel call to trigger functionality residing within another component. This gives additional overhead compared to having components that directly address each other, but is more dynamic.

To ensure the proposed qualities and an efficient engine, Thorn suggests designing and implementing game engines by following the principles of *Recyclability, Abstractness, Modularity, and Simplicity* (RAMS). It should provide *Recyclability* in the terms of reusing available resources. An example is that multiple instances of an entity in the game can share the same memory for the same properties, instead of having duplicates in memory. This can potentially enhance performance without utilizing additional resources. An engine should provide *Abstractness* and *Modularity* as explained with the engine components. And finally, provide *Simplicity* through keeping things as simple as possible, also in the terms of components: as few as possible [69].

All of Thorn's principles support Gregory's game engine definition, and they both share the same vision of a well implemented game engine: A modular and dynamic piece of software which can be used to develop a vast variety of different games. An abstract engine architecture that is designed as a framework or API can be utilized to achieve such modularity [25, 69]. In the end, good game engine design is all about good system design.

Chapter 3

Games in Computer Science

3.1 The Early Ages of Video Games

There is no known documentation proving the very first interactive digital game created, but it was probably developed some time between 1940 and 1951 as it was the first decades of digital computers. The reason is the vague definition of the term, and because available technology could allow undocumented electronic games to be developed as hobby projects and outside of academia. One may claim that a specific game is the first, but it can't be proved, as it is impossible to know if someone developed something earlier that may be categorized as a video game.

The earliest know and well-documented electronic game is believed to be Thomas T. Goldsmith Jr. and Estle Ray Mann's game, patented as "Cathode-ray tube amusement device" - US patent US2455992A [24]. A purely hardware (electronics) game, made for fun and science. It allowed the player to control a missile on a World War 2 inspired radar display. The goal was to shoot down simulated planes within a specific amount of time [24].

The very first proper digital video games appeared in the early nineteen-fifties and were developed by some of the first computer scientist as tools for topics within computer science, such as *artificial intelligence* (AI). Alexander Shafto "Sandy" Douglas was a professor in computer science whom created the earliest known digital video game that displayed graphics on a monitor. It was called "OXO", a *tic-tac-toe* game developed as a part of his PhD thesis [76]. Inspired by known games of play, the digital video game was born as a child of CS.

Video game development remained within the field of computer science and as university hobby projects and experiments until the commercialization and birth of the video game industry in the 1970s. It marked the birth of the first *arcade* game, *Pong*, and the first successful home console. The games were not that complex at the time, and were usually developed by a single person. Games were still developed by scientists during these years, but it marked the beginning of the important separation of games for science and business.

University hobby projects blossomed during the decade and university mainframe computers were often used for video game development. The 1980s was the golden age of arcade games while the console market almost died, but it was saved by *Nintendo*'s *NES* in 1985. The 1990s was a decade of innovation within the industry that led to the huge multi-million dollar game industry we have today. It made 3D graphics the standard and released home consoles so powerful that the arcade industry eventually died [43].

Since then, both the popularity and demand for video games has exploded with an insatiable audience demanding more content and more technically advanced features for every major video game release. It has resulted in one of the world's toughest industries with high development costs and high risks for failure. The industry is thus a technology promoter that keeps pushing hardware and software to the limit in order to satisfy the ever-demanding market. Something that is still true today, and will probably continue to grow together with future advances in CS. It is likely that the development time and costs of a single modern video game bypassed the resource capabilities of most CS projects already in the early 1990s.

There has always been a difference with the primary purpose of game development in the commercial industry and academia. The game industry is all about making money by publishing games that keeps satisfying the ever-demanding market. Modern scientists use video games as a tool in their research (such as Petlund et al. [48], Raaen et al. [50]), as well a topic of research (such as Claypool & Claypool [12], Eisert [46], Waveren [73]).

Game developers prioritize rapid creation of content and the implementation of a large system, while the computer scientists prioritize investigating and researching individual components of the larger system. Scientists do not have the resources to develop a complete video game of commercial quality, while game developers rarely publish any results of their development and research. This indicates that cooperation between the industry and the researchers is beneficial. The industry can provide scientists with the source of state-of-the-art commercial games, which they can use in their research. The research will likely provide beneficial solutions for the games, which can be shared with the industry to improve their future releases.

3.2 Modern Research

The history of video games reveals that the primary goal of the very first scientifically developed games was the development of the game itself. It was to prove that they were capable of utilizing modern technology to create an interactive entertaining experience that illustrated concepts of human-computer interaction [76]. Looking at more modern research illustrates that games have become a topic and tool for research, rather than the goal.

Games have proven to be useful tools for a range of various sub-topics within CS, as well as educational research, and even human psychology.

Other game-utilizing scientific topics are; game case studies, algorithm efficiency, 3D data simulations, artificial intelligence, hardware utilization, and computer networks. Little research is available on the use of games in CS [29], which probably makes it impossible to cover all specific usages of video games in research. However, this section aims at covering the most common and relevant uses for this thesis, all by looking at existing research.

3.2.1 Educational Games

In educational research, games are studied as potential learning platforms and educators. A game is scientifically either modified or developed to exploit its element of fun to convey a specific curriculum. Something that goes hand in hand with Koster's theory of games as entertaining educators [36] A recent example of this is a game called *CodeSpells*, developed at the university of California, San Diego. The game simulates a world with gnomes that have lost their magic abilities where the player, a wizard, shall help them regaining their powers by writing spells in the *Java* programming language. User tests of the game shows that several children managed to learn many of the fundamental concepts of *Java* only within an hour of play [45].

Another example is *Kahoot!*, a digital social classroom game which works like a game show. A user-friendly backend is used by the lecturer or teacher to assemble a collection of questions. The questions can then be presented in class through the *Kahoot!* framework. Participants reply to the questions through their own digital device, and the final results are displayed for the class. This combination of competitive gameplay and student interaction is believed to increase the students' overall learning outcome. *Kahoot!* is created and maintained by professor *Alf Inge Wang* at the Norwegian university of science and technology (NTNU) [63].

3.2.2 Psychology

As explained in section 2.2, video games are illusions of real-time worlds making them interesting platforms for interdisciplinary research of psychology and CS. This primary concerns the human perception of quality, where psychology may be used to find requirements or upper boundaries in order for a game's components to be credible. This research covers topics as frame rate and latency sensitivity, game AI credibility, and the quality of experience (QoE). An example is the previously discussed requirement of a 60 FPS game loop, which ensures the real-time requirement of a game system.

Mark Claypool and *Kajal Claypool* are two computer scientist that have made several publications within this topic, investigating network latency in various types of games [11, 12]. They have built latency test-beds with real human players where they use middleware to adjust the latency of a game and study the effect player's performance. Several of their findings are summarized in their publication entitled *Latency and player actions in online games* (2006)[11]. Their results show that games with a

Model	Perspective	Example Genres	Sensitivity	Treshold
Avatar	First-person	FPS, Racing	High	100ms
	Third-person	Sports, RPG	Medium	500ms
Omnipresent	Varies	RTS, Sim	Low	1000ms

Table 3.1: Latency thresholds for various games. Copied from Claypool & Claypool (2006) [11].

single user controlled avatar (entity) are the most sensitive with a latency threshold of 100-500ms, while games with multiple controllable avatars have a threshold of 1s. High-paced action games, such as first-person shooter and racing games, are the most latency-intolerant. This is not a surprise as they are games requiring a high update frequency to be perceived as running in real-time. Details of their findings are available in table 3.1 [11].

3.2.3 Game Case Studies

A game case study is the investigation of an existing game in an attempt to learn and document some specific functionality. It may reveal smart solutions to how features have been implemented which the researchers can use to their advantage. They may find smart and efficient solutions that can be used to solve problems in their own research. Alternatively it can be for the sole purpose of documentation in order to learn how to work with the game. An example is *Quake III Arena game structures* by D. Stefyn, A.L. Cricenti, and P.A. Branch (2011)[21] where they analyzed the source code of Q3A to document its network communication system and data structures.

Fabien Sanglard has published many technical reviews of various video games, including Q3A. They are all available for free on his personal website and all highlights creative and excellent solutions in the various games [56]. The reviews are not academically presented, but are good game case studies.

A larger part of this thesis may also be categorized as a game case study, as I had to gain deep knowledge of the *ioquake3* (*idTech 3*) engine in order to be able to perform the implementation of virtual clients.

3.2.4 Algorithmic Development & Optimizations

As discussed in 2.2.1, video games are software that continuously performs heavy calculations that need to be performed fast to reach their real-time deadlines. This makes games a well-suited target for optimization and algorithmic development. Modifications to a time consuming function in a game can yield significant performance differences. Algorithmic optimization was formerly a hot topic within graph traversal for *pathfinding* (see section 3.2.5) in artificial intelligence (AI). Several topics of AI apply to both games and robotics [34], but games are usually the cheapest, thus being good development and testing tools.

An example is Peter K. K. Loh's and Edmond C. Prakash's (2009)[39] publication on *Performance Simulation of Moving Target Search Algorithms* where they utilize a small prototype maze game to compare existing *moving target search* (MTS) algorithms with a newly developed *abstraction MTS* algorithm. It turns out that abstraction MTS is performance-vise competitive and scales better than the other algorithms [39].

3.2.5 Artificial Intelligence

AI is a cross-disciplinary field that applies to games and academic research. The concept is however the same, which is to make computers capable of thinking and to make their own intelligent decisions based on available data [42]. This section will only scratch the surface of game AI, as it is not the primary topic. AI research can be specifically aimed at games, but games can also, as previously mentioned, be a cheap substitute for robots. The primary difference of game AI compared to the other disciplines is the ability to fake intelligence, as it only needs to be perceived as intelligent. Adding more AI to a game than needed can actually have negative effects [42]. Other AI disciplines require the AI to be as good as it can be, because one can't fake intelligence in true AI.

Terms of AI

Games simulate worlds of virtual objects such as rooms, walls, characters, weapons, projectiles, etc. These objects are often referred to as *entities*, a term used to describe every virtual object that intelligent entities can interact with, including themself. The proper AI term for these intelligent entities is *agent* and includes both human and AI controlled entities. AI controlled entities can also be separated from the human entities with the terms *bot* (AI) and *player* (human). A bot is an abbreviation of robot, which makes sense, as they are computer-controlled entities just as real-life robots. Some of the most common topics within game AI are decision-making (finite state machines & decision trees), movement (steering & pathfinding), and machine learning.

Decision-making

Also known as *agent reasoning*, is all about making the agents understand the rules of the game in order to make decisions which is best for them in their given situation. A bot in Q3A should for instance know that it is wise to locate a gun and ammunition before it decides to fight an opponent. The bot retrieves some information as input, and outputs a game action. The input should provide the same information as a real human player would get, including the state of the game, possible actions and their outcome, visible entities, team mates, health, map triggers (e.g.: jump pads, door switches), etc [42]. It is thus important that engines should provide an easy interface for adding bot information in the level editor and other game tools.

As for the output, the bot will need a decision-making system to find the best action based on the available data. Most common in games are the implementation of *state machines* and *decision trees*. *Neural networks* may also be used for decision-making, but they are usually avoided due to their high complexity that makes them hard to get right.

State machines are good for situation dependent dynamic agent behavior. A typical implementation allows each bot to only have one active state at a time, which can be changed through transitions. Each state performs its state-specific action and evaluates the current state of the agent and its surroundings. The result decides whether it should transit to a new state or not. This specific type of state machine is known as a Finite State Machine (FSM), as there are a finite number of states available [42]. The following is a simplified example of a bot in Q3A. The bot's initial state can typically be *hoarding*, where it needs to gather weapons, ammunition, and body armor. If the hoarding state notices that the bot is combat ready it will transit into *seek* state, where it navigates through the world looking for enemies. The FSM will ensure that the bot continues to perform rational decisions until the game session ends.

A *decision tree* is an attempt to organize knowledge (the input) to create proper corresponding actions. It is a binary tree implementation with a root node that checks whether a given condition from the input is met. The binary result moves on to the corresponding child node that evaluates another condition, and continues to traverse the tree until it reaches a leaf node. Each leaf node decides a final action [42].

Decision making in Q3A is implemented through a network based FSM which utilizes *fuzzy logic* to make transitions. Fuzzy logic uses a weight, a numeric interval (e.g.: 0-100), where fuzzy values are evaluated in comparison to each other and the heaviest weighted value is selected. The actions within each state are handled in a decision tree like manner [73].

An example of decision-making in research is A. Braun's, B. E. J. Bodmann's, and S. R. Musse's publication *Simulating virtual crowds in emergency situations* (2008)[7]. They developed a game-influenced data driven system to simulate the behavior of crowds in emergency situation, utilizing FSM with psychological factors to trigger transitions.

Movement & Pathfinding

Movement & pathfinding is about the AI required to navigate and steer through an (virtual) environment. It requires that the bot has geometrical information about itself and the environment, which enables it to move naturally through the environment. Pathfinding, also known as *path planning*, is a term closely associated with movement. It is the use and development of algorithms that evaluate and finds available paths in a graph-defined environment, which has several advantages compared to fixed routes. The primary one is the support for dynamic routes, which adds support for re-routing around obstacles and broken paths.

It also provides automatic movement support in new engine-compliant maps. Secondly the bots become more realistic because they adapt to their surroundings and won't necessarily always pick the same routes.

There are several known pathfinding algorithms, but the most common are *A** (pronounced *a star*) and *Dijkstra*'s algorithms. They are both graph traversing algorithms and are thus not compliant with geometric data of a simulated three-dimensional world. The solution is to build a simplified graph version of every map which the bots can traverse with the algorithms. Engine tools like map editors will need to include support for path specification and auto-generation of graphs. Weighted graphs can be used to specify the cost (time-wise) of each path, allowing the bots to find the fastest route to a location.

Traversing graphs to find the fastest routes are quite computationally heavy tasks, and it needs to be efficiently implemented for the games to remain within the real-time deadline. These algorithms do however not scale very well in huge environments, such as the worlds in massively multi-player online games (MMOGs). A known solution is to split the graph into a hierarchy and to utilize a step-based approach to traverse the graph. This is known as hierarchical pathfinding [42].

As explained, pathfinding requires efficient graph traversal that makes research on network routing and graphs apply to game AI as well. An example is X. Liu's and D. Gong's publication on *A Comparative Study of A-star Algorithms for Search and rescue in Perfect Maze* (2011)[38] where they evaluate three variants of hierarchical A* algorithms against each other and the simpler *depth-first* algorithm. This is done by making a video game like simulation of a *perfect maze*[1] that is solved by a bot utilizing the different algorithms. The results show that A* outperforms depth-first in most situations, and that the three individual A* star algorithms have different advantages in the various mazes [38]. This can be useful for game developers as they can find the best A* variant for their maps by comparing them with the different maze characteristics.

Other examples are A. Sud's, E. Andersen's, S. Curtis', M. Lin's, and D. Manocha's publication on *Real-time path planning for virtual agents in dynamic environments* (2008)[66], and previously discussed Braun et al. (2005)[7]. Both contribute to AI agents' movement in crowd simulations.

Q3A has implemented pathfinding and movement as a part of the *Area Awareness System* (AAS). It is a component of the bot system that contains all information about the world's current state, including entities, paths and routing. The AAS data is a special 3D representation of the world that is formatted and preprocessed for efficiency. Maps provide AAS system with navigational waypoints that are organized into *areas*. Areas are clusters of waypoints that are generated by locating connected waypoints with minimal navigational complexity. Navigation opportunities between areas are calculated through so called *reachabilities*. These are only added if a bot can easily travel between two areas. This navigational data is pre-calculated and stored in a routing cache for each map. The cache sorts

[1]A maze without loops.

connected areas into clusters that are separated by special areas known as *cluster portals* [73].

Waveren (2001)[73] decided to pre-calculate routing information in a cache because it is common that maps consist of 5000 areas or more, making the Dijkstra and A* algorithms too slow for real-time traversal. This is good for efficiency but requires more memory. The cached routing data is traversed in real-time using a simple *breadth first* algorithm [73]. A more efficient real-time solution could probably be to exploit his hierarchical cluster separation, utilizing a step-based A* algorithm for hierarchical pathfinding.

Machine Learning

Also know as *learning AI*, is a term describing AI that dynamically adapts its behavior based on virtual experience. It is an attempt to make AI more like humans, as we use our knowledge and experience when making decisions. A bot can for instance analyze and store data on an opponent's various patterns, such as movement and fighting tactics. It can then use on this data to anticipate the opponent's actions, making the bot a greater challenge and its intelligence more credible. This is known as *online learning*, where the bot adapts in real-time. *Offline learning* allows the bots to see the greater picture, where it can learn from data on previous games to develop better strategies and get better at making decisions [42].

Learning is currently not commonly implemented in games as it is difficult to implement properly, and because AI without learning is in most cases perceived as sufficient. The complexity of implementation depends on the required level of intelligence. It may be as simple as tweaking numbers, and as complex as the utilization of *neural networks* [42]. Artificial Neural Networks (ANNs) are modeled after the human brain, with the goal of simulating true AI; that is AI capable of making decent decisions based on its experience. Research on ANNs have proven that they are good for artificial learning, but no one have so far been able to create true AI with them [52]. In fact, a recently published article by P. Maguire, P. Moser, R. Maguire, and V. Griffith (2014)[40] claims to have mathematically proven that we will never be able to create true AI, as it would require functions that are non-computable.

3.2.6 Visual Interactive Simulations

Simulation is a vague term that applies to multiple cross-disciplinary fields of research and industry. This section will only focus on the fields relevant to games. Section 2.2 discussed that most video games can be considered interactive simulations with visual output. Visual interactive simulation (VIS) is a term first mentioned in R.D. Hurrion's PhD from the *University of Warwick* in 1976 on scheduling problems in manufacturing systems. His work on developing manufacturing system simulations unveiled that most manufacturing systems had a real human actor with some control of the entire process. This is difficult to reproduce in a simulation because the human actor's decision patterns. The solution became interactivity that allows a real human to control the simulation's output in real-time. Research on VIS continued at the *University of Warwick* in the following years with much industrial cooperation, including *Imperial Chemical Industries* (ICI) and Rolls-Royce [4]. The primary difference between VIS and video games seems to be the software's objective. Video games aim at entertainment that allows unrealism, while VIS should be as accurate as possible to ensure realistic output and proper education.

VIS is widely used in research and industry as of today, used as educational tools for staff training and simulation of various processes. This includes, but is not limited to medical research, and military, space & maritime training simulators. Roger D. Smith published a tutorial on military simulations in 1998[61] stating that governmental investments in military simulations are increasing. This seems still to be true as *Kongsberg Defence Systems* recently signed 49 MNOK contract on upgrading the Royal Norwegian navy's *PROTEUS* simulator infrastructure [67]. Smith [61] also presents the history of military simulations, which shows that the need for war simulation origins in war games. Military simulations are often real-time as they aim at training personnel through realistic combat scenarios.

The research and work on VIC lead to a *IEEE* standard for *Distributed Interactive Simulation* (DIS), first released in 1998. It is a standard that applies to simulation of warfare, logistics, management, environments, radio communications, environments, and more [32].

> Distributed Interactive Simulation (DIS) is a government/industry initiative to define an infrastructure for linking simulations of various types at multiple locations to create realistic, complex, virtual worlds for the simulation of highly interactive activities. This infrastructure brings together systems built for separate purposes, technologies from different eras, products from various vendors, and platforms from various services and permits them to interoperate. DIS exercises are intended to support a mixture of virtual entities with computer-controlled behavior (computer-generated forces), virtual entities with live operators (human-in- the-loop simulators), live entities (operational platforms and test and evaluation systems), and constructive entities (wargames and other automated simulations).
>
> — *IEEE Std 1278.1a-1998 [32, p. iii]*

There are also other types of non-interactive simulations that utilize technologies relevant to video games. The lack of interaction prevents them from being dynamically altered during run-time, but it is common to make them configurable through configuration files or scripts. Such simulations have been used for research on environmental disasters & human crisis situations to anticipate the outcome of various scenarios. An example is Braun et al's (2005)[7] configurable simulation tool of crowds in emergency situations. A XML script can be configured to simulate different crisis scenarios. This is used to simulate and learn how crowds behave in different emergency situations. Another example is Brodtkorb, Sætra and Altinakar's configurable GPU based shallow water simulation (2012)[8] which can be used to create a flood simulator that predicts affected areas [51].

3.2.7 Hardware Utilization

Games have been used to prove that better utilization of modern hardware can dramatically enhance a system's performance. Common are the utilization of *Single Instruction Multiple Data* (SIMD) processing and heterogeneous multicore architectures. Brodtkorb et al's (2012)[8] publication provides a brief historical overview on the use of general purpose GPU (GPGPU) programming in CS. It started over a decade ago as simple tests to prove that the GPU could be used for general purpose programming, and developed into standard equipment of modern super-computers [8]. This is possible due to the unique parallel GPU architecture that provides hundreds of small processing cores and an advanced memory hierarchy. It makes GPUs ideal for mass calculations such as graphics processing and video encoding [18]. Brodtkorb et al's (2012)[8] flood simulator uses GPU to perform its heavy calculations, making it capable of accurately simulate the first 66 minutes of the 1959 *Malpasset* dam break, which is 400 times faster than some of the existing computing approaches [51]. A more game related example is Sud et al's (2008)[66] study on crowd pathfinding, where they utilize the GPU to perform real-time pathfinding of multiple agents in a virtual 3D environment.

GPUs are probably the most common hardware utilized to accelerate high-performance computing as of today, but research has been done on the utilization of other hardware as well. Multi-core CPUs were embraced by scientists when they emerged at the beginning of this century, leading to research such as Tulip, Bekkema, and Nesbitt's (2006)[70] publication on *Multi-threaded Game Engine Design*. It states that most games where single threaded at the time, and that next generation game engines must embrace concurrency to benefit from new multi-core hardware. The publication documents game engine concurrency issues, and analyzes parallel opportunities. They propose that the issues can be solved by hierarchically structuring a game engine's tasks into a *task tree*, which is to be processed by a *thread-pool* [70]. A more detailed example is Raaen et al's (2012)[50] research on *LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition*. They utilize concurrency on a game server, by designing and implementing an almost lock-free, thread-pool based architecture (LEARS) that support hundreds of clients in one scene [50].

More uncommon architectures have been used as well, such as the *STI Cell Broadband Engine* (Cell), which is the unique asymmetric multi-core processor used in *PlayStation 3* (PS3). It is a nine-core architecture consisting of one *Power Processing Element* (PPE), and eight Synergistic Processing Elements (SPEs) [64]. The primary purpose of the PPE is to manage the SPEs, indicating that modern high-scale PS3 games need to utilize the multiple cores in order to run smoothly. Stensland, Espeland, Griwodz and Halvorsen (2010)[64] analyzed 14 different *Motion-JPEG* video encoder implementations and performed related experiments on both Cell and GPUs to identify optimization opportunities on both platforms. Video encoding is of high relevance to games in terms of *cloud gaming*, see section 3.2.8 for full details.

3.2.8 Networking

In terms of networking, online video games are considered *interactive thin-stream applications*. They are thin stream as they usually transmit streams of small data packets with high intervals, compared to *greedy* streams which goal is to transmit larger amounts of data as fast as possible [47]. It is given that only network based online games with some sort of distributed processing (e.g. a central server) is relevant for network research. Game servers are usually considered more interesting than clients, as servers are the heaviest loaded component of online games.

Latency and QoS

The previously discussed latency requirements of Claypool & Claypool (2006)[11], listed in table 3.1, proves high-paced action games and role-playing games (RPGs) as interesting network applications because they completely rely on high packet throughput in the unreliable best-effort Internet. An example is Petlund's PhD thesis (2009)[47] where he analyzed packet data of three different games to verify a thin-stream TCP issue and to prove that his proposed solution resolves the problem.

Quality of service (QoS) is a hot topic within networking that attempts to improve the Internet with specific promises of reliability and speed. It proposes a new revenue model for Internet service providers (ISPs) where they can sell subscriptions of differentiated qualities. They may for instance provide a *premium* subscription with the highest reliability to their most demanding customers, such as heavy online gamers. Games are due to their real-time demands of high relevance to QoS research. An example is Armitage & Zander (2004)[3] empirical study on online network games' QoS requirements. Their contribution is a proposed list of requirements that needs to be met in order to satisfy the gamers.

Cloud Gaming

More recent research indicates that increasing popularity of *cloud computing* and cloud services have made the requirements on QoS even greater, as a new trend has emerged in both the industry and in academia; *cloud gaming*. The idea is to provide externally processed game services, eliminating the end user's need for expensive and specialized gaming hardware. A client forwards the player's input to the server, which processes it and continuously transmits the game's video output to the client. The downside is the requirement of a stable and continuous HD video stream that currently faces great challenges in terms of latency. It completely alters the QoS requirements in comparison of traditional gaming. An advantage is that games become platform independent lightweight client software, potentially allowing the user to play any game on any device. It will also require every client to have a valid subscription to the specific service, eliminating the possibility of video game piracy [33].

Cloud gaming's primary bottleneck is the bandwidth consuming video stream, and the development and use of video encoders is the primary concern. HD video encoding is a computational heavy task that is added on top of the already heavy game engine processing, making it even more cumbersome to meet the real-time demands. It poses several new challenges that have yet to be solved, but we might get there gradually. *Valve Software* just released a cloud gaming-like solution within their *Steam* platform, allowing one to remotely play games from another local computer in the network [41]. It indicates that the current fast local network technology is sufficient.

Network Load Generation Tools

Yet another topic within networking is network load generation tools, also known as traffic generators. Modern online software systems are targeted at the simultaneous use by thousands, and even millions connected users. This poses a testing challenge as neither academia or the industry will usually be capable of performing system load testing of such scale with live sessions involving real people. Automated load testing of AI controlled traffic generating virtual users, or virtual clients, is required to test their systems under various loads [60]. These are tools that allow you to specify a number of virtual clients to perform a specific procedure. This type of testing is sometimes referred to as *stress-testing*.

Load generation has been an active research area for specific types of servers, and several industry standard tools have been developed to stress-test file servers, web servers, and similar. Common for these is that they are customizable and scalable tools that generate user-authentic traffic. Modern games would also benefit from good load-generating tools and the industry develops such custom tools for their games. [60]. The primary challenge of developing general load generation tools for games is that every game is different. In order to generate representative traffic, a load-generating tool needs to be aware of essential game information such as map data, game logic (rules), movement, and network protocols.

Very little scientific research is available on the topic, but the little that exists seem to have adapted the concepts of other server software's load generation tools in an attempt to develop an industry standard all-purpose game load-generating tool. First is Jung, Lim, Sim, Lee, Park, Chung, and Lee's (2005)[35] *Virtual Environment Network User Simulator* (VENUS) system. A software suite developed with the goal of becoming the standard model for online game testing. The publication shows that their system can be configured to stress-test an online game's authentication server (login procedure) and make virtual clients move to specific positions afterwards. They state the components of their suite can be easily configured to support any game, but provide no instructions or examples on how they do it [35]. However, their next publication revealed that the game's client logic had to be incorporated into the VENUS system for it to work [10].

Five years later some of the same authors published documentation on an improved version of the tool; *VENUS II*. Cho, Sohn, Park, and Kang (2010)[10] presents an updated software suite that seems much more configurable and easy to use. The suite consists of a packet capturing tool, a packet analyzing tool, and a virtual user control tool. All tools provide a user friendly GUI. In order to use VENUS II, you must first use the packet capturing tool to capture network traffic from a live game session with real players. This is stored into packet database that is utilized by the packet analyzing tool. It interprets the packet data and generates data on the *game map* and the *game grammar* (rules). The packet analyzing tool is configurable and should be configured for each game for best results. The virtual user control tool utilizes the generated game map and grammar to create and control both single and groups of virtual clients. The virtual clients are capable of logging in, navigating the world, and attacking [10]. Downsides of the solution are that packet capture and analysis needs to be done for every single map and that the clients lack AI for gameplay. The produced network traffic is thus not representative for a real player. Other issues are that most of the tools are in Korean, and that there are no traces of the actual VENUS tools online.

Some other Korean researchers were inspired to propose a similar tool, but with a slightly different concept. Shin, Kim, Sohn, Park, and Choi's (2010)[60] uses their previous research on game traffic packet analysis to propose a system that analyses player behavior and patterns (learning AI) to generate reliable packages simulating a proper working virtual player. They evaluate their system by manually ordering and comparing the data from the packet analyzer and claims the results indicates that they are capable of creating traffic similar to real players [60]. Their theoretical proposal may seem promising, but it is impossible to tell how well it will work before they develop a working prototype. No evidence of further progress on this system has been discovered. This is however difficult to find since they did not name their proposed system.

All the proposed network traffic generating systems have proposed interesting concepts on the development of a general online game load-generating tool, but none are completed or working. I am especially concerned about how these systems handle compressed and encrypted game traffic since most modern games, including Q3A, both compresses and encrypts their network data. None of the publications mention anything about it. Also, all online games are unique pieces of software with huge differences. So the entire idea of creating such a system is vague. It is as complex as creating a load generation system that is capable of generating user-authentic traffic for all server software. This is also reflected in the complexity of the two VENUS systems. It is likely that it will take less time to create a custom virtual client implementation in a specific game, than it will to configure any of the VENUS systems to produce representative data.

3.3 The Need for an Open Load Generation Tool

This thesis has so far discussed the definition of video games, their inner workings, and scientific uses. It shows that video games have always been, and currently are, highly relevant within various topics of CS, both as topics and tools. The reason is the demanding nature of video games with high requirements to hardware, software and network. Commercial video game development advances with technology. Research should thus ideally work with modern state-of-the-art video games, so that scientists can work on software pushing the available technology to its boundaries. The problem is to find such a game.

Section 3.1 showed that commercial actors most likely bypassed the game development capabilities of CS projects two decades ago, due to the different priorities in academia and industry as discussed in section 3.1. It means that researchers will need to gain access to commercial games or find open source game projects in order to find games of commercial quality. Open source games are however not ideal candidates. My research, documented in section 4.1.3, unveils that known open source games lag behind the commercial due to the voluntary nature of open source projects. In fact, several open source projects utilize engines of old commercial games that have been open sourced years after their initial release [29].

It is possible to use commercial games in research, but this is problematic. They are rarely available in source code form. New, the code is considered a company secret while releasing older code is either ignored or can be too costly. Developers are sometimes willing to co-operate with researchers, but will often limit how much of the inner workings can be disclosed. This makes the research cumbersome as well as difficult to reproduce, potentially decreasing the quality of the research material.

A third alternative for the researchers is to develop their own prototype games to use in their experiments. Examples include Liu et al [38] and Raaen et al [50], which both try different concepts in simplified prototype environments. Raaen et al [50] states that their specialized prototype is a weakness in itself, as it lowers the research's credibility compared to a commercial quality implementation. One can assume that it is the same for Liu & Gong [38], Valente et al [72], and other similar cases. This research should ideally be implemented and tested in complete games in order to achieve more realistic results. Ideas need to be embedded in a full-scale production in order to get fully realistic results. There is thus a need for a universally available commercial quality video game that can be freely used for scientific purposes.

Even though they are behind the industry, stable released open source video game projects are potential candidates for implementing scientific concepts in a large-scale system. They need to be evaluated and most likely adapted in order to serve as a test bed for scientists. The evaluation is required to determine if a game is stable, if it is representative for the industry, and whether it is computationally heavy enough to measure the effects of implementation changes.

Identifying potential games is just the first step. A game will most likely need modifications in order to measure high-scale utilization of the game. How depends entirely on the type of research, but the focus of this thesis in networked games. Network and server load can be generated through the use of live sessions with real people as test candidates, such as Claypool & Claypool [11]. It can alternatively be done through automated AI controlled virtual clients, such as Raaen et al [50]. The latter is beneficial as it allows experiments to be done by a single scientist requiring little time and hardware. A live session will typically require a lot of planning, multiple hardware resources, facilities, real test subjects, and several people to assist.

An online game with automated AI-controlled virtual clients is an ideal tool for network traffic and server load generation, as it is purely automatic and generates realistic traffic. This does not automatically include online games with bot support, because a bot is not necessarily considered a virtual client. A virtual client must run on a client machine, perform similar actions as a regular human player, and transmit authentic network traffic to a server. A game implementing this functionality is not only a game, but also a load-generating tool that can be used to measure game and network performance. It is a good test-bed for research on networks, servers, AI, and probably also other topics within CS.

3.3.1 Project Statement

The evaluation of modern computer science indicates a need for open source game projects of commercial quality. This is required to achieve high utilization of the latest technologies. Finding such games is however difficult because most open source projects lacks the resources to build games of commercial quality. In fact, many open source projects are built on older commercial game engines. They are in other words far behind the industry, and will most likely never be able to keep up. Researches could settle with commercial games instead, but it is difficult due to corporate publication restrictions that may significantly reduce the credibility. The best solution is thus probably finding a recent open source project that is close to commercial quality.

This thesis aims at identifying modern open source games that are representative of commercial quality games. Open source games will be evaluated against a specific list of features and qualities that are considered common requirements by computer scientists. The list will be developed by evaluating data from a quantitative survey aimed at computer scientists. This research may benefit all relevant fields of computer science. The results enable researchers to easily select one of the proposed games, or use the evaluation criteria to find open source projects on their own.

Selecting a game is the first step to a useful tool. Implementing support for virtual clients allows scientist to efficiently perform authentic large-scale testing without significant amounts of resources. This thesis converts an open source video game, *Quake III Arena*, in to a user-friendly load-generating tool with virtual clients. Q3A is a relatively well-designed game engine, and implementation has been done according to Thorn [69] and Gregory's [25] guidelines for good engine design.

Part III

Implementation

Chapter 4

Finding an Appropriate Game

4.1 Establishing a Foothold

The decision to use *Quake III Arena* for the implementation is not arbitrary. This thesis' foundation is built on research identifying the need for a freely available video game of similar or close to commercial quality, which can be used as a tool for research. The first part of the thesis was to find such a game, and there was previously no known research on the topic. Commercial games are not used due to their restrictions, making open source games the only candidates [29].

4.1.1 Identifying Potential Games

Finding open source games online is easy and requires no more than a simple web search, or a peek at the dedicated *Wikipedia* page on the topic [75]. Identifying games that satisfy the common needs of computer scientists is however difficult, as they need to be evaluated against a list of features and qualities of a game. The problem is that no such list exists, and there is no available research on the topic [29]. It is impossible to evaluate any game without clear requirements, so I decided to identify a list on my own by gathering data from reliable sources; computer scientists.

The first step was to interview one of my supervisors with industry experience, *Kjetil Raaen*. It provided the foundation to develop a list consisting of 60 different features and qualities that may be implemented in video games. It covered most of the currently known technical features that are considered realistic to implement in games. Both the basics, such as user input through well known devices, and less common features, such as voice control was covered. The survey concerned software, and assumed that all necessary hardware and drivers were in place and working properly for the various features to be fully functional. The list was built into an online survey, where the participants were to categorize each feature or quality as either *unimportant*, a *research requirement*, an *engaging requirement*, or *both*. An *engaging requirement* is a feature that is needed for a game to be engaging for potential test subjects.

The survey was active for approximately a month, and invitations

Feature	Requirement %	Unimportant %
Performance	91	5
Logging System	77	18
Latency Intolerant	68	18
Simple Physics	68	23
3D Support	68	14
Technical Documentation	68	5
Simple AI	64	27
2D Support	63	14
Adaptability	59	23
Scripted AI	59	23
Good Code Quality	59	18
Computational Heavy AI	55	32
Reliability	54	27
Player vs Environment (PvE)	54	23
Large networked (13-64)	54	36
TCP Support	50	32
Single Player Mode	50	45
Player vs Player (PvP)	50	27
Massive Networked (65+)	50	41

Table 4.1: Top Evaluated Game Features & Qualities

to participate were sent to computer scientists that previously had or currently were interested in using video games for their research. Most of these researchers were based in Norway, among them some scientists from *Simula Research Laboratory*, *SINTEF*, and *The University of Oslo*. It was also sent to various researchers abroad. The survey had a total of 22 participants, which reflects that few computer scientists replied that they felt their experience and work was relevant to the survey. Participants from the game industry were not invited, because the problem discussed is unique to academic research. The resulting list of features and qualities are presented in table 4.1.

Survey results were used to find good candidates by using them in comparison with open source games from *Wikipedia*'s vast list [75], games used in previous research, and open games known to me and my supervisors. Games from Wikipedia's list with most recent update prior to 2012 were considered inactive and therefore excluded. The games examined more thoroughly were *Doom 3*, *OpenArena*, *Vega Strike*, and *PlaneShift*. Each game was evaluated with the following steps.

4.1.2 The Evaluation Process

The first phase of the data gathering process involved installing and playing the respective games. It is a quick and an effective approach to get an overview of each game, indirectly providing information on various game mechanics, its quality and whether or not it is a stable product.

Graphics quality is evaluated visually while playing the game. This gives an immediate indication on the quality of the graphics, and whether more advanced graphical features, such as *shaders*, are supported or not. The approach also reveals implemented user interface mechanisms, such as game menus, buttons, in-game chat and similar.

Some information on embedded physics was retrieved by interacting with the games in ways that might not have been considered by developers. This involves jumping off various ledges, firing weapons at objects not usually considered targets, and experimenting with various game mechanics. Such gameplay reveals much about interaction between objects in the game. Looking at the contents of various menus and game settings can also retrieve other important information. Setting up a local server will for instance let you know how many interacting concurrent players the server supports.

The second phase was retrieving information by collecting data from official websites and documentation available for the games and engines. Primarily to find technical specifications revealing features either implemented in the games, or *hidden* engine features not used by the games. This can reveal a significant amount of features and qualities not visible by simply playing the game. It is also more efficient than attempting to interpret features hidden within the source code. The amount and quality of the documentation is highly variable. Because open source games require extensive cooperation between geographically distributed developers, more documentation is usually available than for commercially developed games [29].

The final phase is to investigate the undocumented features by studying parts of the source code. This requires obtaining a copy of the source code from the official version control repository or file server, and opening the files in the intended development environment, if any. Some will immediately compile, link and run successfully, while others require the retrieval of game data or additional resources. For games with commercially licensed data, researchers need to purchase or otherwise legally obtain these files. These are not needed to analyze the source code, but can make it easier by allowing you to navigate the code through testing and debugging.

A good start is to obtain an overview of the source code structure in order to find out where to look for various potential features. Analyzing the code reveals code standards, features, and the code quality. It can be a time consuming task, we were only interested in examining the code for specific functionality and overall quality, which does not require examining every line of code manually. Once done, final in-depth searches for the unknown features are done, involving basic file search and reading. Some features are easier to determine than others, due to their complexity and nature. Features are marked as they are found or declared missing.

Feature	Doom 3	OpenArena	Vega Strike	PlaneShift
Performance				
Logging System	✓	~	✓	✓
Latency Intolerant	✓	✓	✗	✓
Simple Physics	✓	✓	✓	✓
3D Support	✓	✓	✓	✓
Technical Docs.				
Simple AI	✓	✓	✓	✓
2D Support	✓	✓	✓	✓
Adaptability				
Scripted AI	✓	✓	✓	✓
Good Code	✓	✓	✗	✓
Comp. Heavy AI	✓	✓	✓	✓
Reliability	✓	✓	✗	✗
PvE	✓	✓	✓	✓
Large Networked	✗	✓	✗	✓
TCP	✗	✗	✗	✗
Single Player	✓	✓	✓	✗
PvP	✓	✓	~	✓
Massive Networked	✗	✗	✗	✓

✓ implemented ~ partially ✗ absent

Table 4.2: Evaluated Open Source Games

4.1.3 Evaluating Potential Games

This section contains an in-depth analysis of the games selected for evaluation. A visual summary of the evaluated game's qualities is presented in table 4.2. It compares the top results from the online survey with the available features of the four selected games. Qualitative features such as performance are not evaluated in the table, but described further in the section for the individual game. The reason is that qualities cannot be represented as binary values.

Doom 3 (idTech 4)

I consider *Doom 3: BFG Edition* and it's upgraded *idTech 4* engine the most complex and technically advanced open source game currently available. Most of the original game's development was done prior to its commercial release in 2004 [23], but some modifications were needed in order to release it as open source under the *GNU General Purpose License* version 3 (GPLv3) by Timothee Besset (user *TTimo*) on *GitHub* in November 2011 [6]. The game was re-mastered and re-released as *Doom 3: BFG Edition* in 2012 with multiple enhancements, some from the *idTech 5* game engine. These additions were also released as open source only a month later [54].

However, only the source code is freely available, and not the binary data files (game assets). The data files can be obtained by purchasing a copy of the game. A custom compiled engine needs to be configured to read these. However, you will not be able to publish new content or updates with the original game data, as this is only available under commercial license [5]. This can be used for research, but the complete experimental setup cannot be shared.

The engine is mainly written in C++, using OpenGL for 3D graphics, and contains 601032 lines of code [55]. The source code defines various shared libraries as well as an executable.

The source code is of high quality with good naming conventions, some good comments and it is relatively easy to read. It contains clear traces of optimization, which is expected, as it is a major production that sold more than 3.5 million copies [58]. This is also an indication of stability, as it has been tested by commercial actors and been played by millions. The developers of *Id Software* have also implemented their own memory manager, or *garbage collector*, for faster dynamic memory allocation.

The code contains some less common hacks, where the most notable one overrides all *protected* and *private* keywords setting them to *public*. This is required by their custom run-time type information (RTTI) functionality.

Doom 3 is primarily a single player first-person shooter (FPS) game, with an online multiplayer mode supporting up to 8[1] players in four different *Deathmatch* modes, allowing free for all fights. However, one of the programmers, John Carmack, has stated that the engine can support more than 4 players but that they decided to restrict it for game design reasons [22]. This can be modified, as the source code is freely available. The game is an intense action FPS experience, implying the need for low response times as suggested by Claypool & Claypool (2010)[12]. This is reflected in the code with built in latency-hiding mechanisms. The latency sensitive section of the network code uses the UDP protocol with reliability mechanisms implemented in the engine.

A natural part of the single player functionality is the need for *savegames*, which is fully supported by the engine. The game was released before the concept of *Cloud storage* went public, so it is not a part of the engine itself but can be obtained by purchasing and running the game through Steam, although not as open source.

The original project found on *GitHub* does currently only build on Windows using *Microsoft Visual Studio 2010* or later, and on OSX using *Xcode 3.2* or later. It is using operating system (OS) specific libraries, such as the Win32 API for various low level operations including threading and network. This makes it somewhat cumbersome to port to other platforms, but some ports are already available as independent *Git* repositories (Besset (2011)[5] and Sanglard (2012)[55]). Threads have been utilized for time-critical functions that should not be limited to the frame rate of the game, including sound mixing.

Adapting the engine or game itself to be used for other game types than FPS is possible as it is open source. The engine has, however, primarily

[1]Original 4 players, the support for 8 players were added in the *Resurrection of Evil* expansion.

been used by FPS games [74], so it is likely that it will require a significant amount of work to modify it into other genres.

There is unfortunately little or no official technical documentation of the engine apart from the source code. An *idTech* developer published an engine modification guide that is useful, though not complete. There is also a very good code review available by Fabien Sanglard [54, 55] but it also lacks some information, including networking. The only thing available close to design documentation is the game manual, but no official documents have been published.

Other available and high quality engine features are logging, physics, support for scripting, artificial intelligence (AI) and a library for graphical user interface (GUI) components. See table 4.2 on page 40 for an overview of the most important implemented features.

OpenArena (ioquake3)

OpenArena is a pure first-person shooter developed using *ioquake3*, an open source community-developed enhancement of the *idTech 3* engine. *idTech 3*'s source code was released as open source together with the game *Quake III Arena*'s source under the *GNU General Public License* (GPLv2) in 2005 [59]. The binary data files generating the content of the game are as with *Doom 3* still closed and copyrighted. *OpenArena*'s intention is to provide a free and enhanced alternative to *Quake III Arena*. *Quake III Arena* is, as Doom 3 a commercial computer game, implying a polished and stable game. The *ioquake3* project claims to have fixed many known bugs in the original engine, patched some security issues and that they have enhanced the overall stability of the entire engine and gameplay.

The engine and the game is primarily developed using C, using OpenGL for rendering 3D graphics. The original *idTech 3* engine consists of 367815 lines of source code [55], which is almost half the size of *idTech 4*. *idTech* and OpenArena are both well recognized and previously used in computer science (such as [73], [46], and Parry (2007)[44]). *idTech 3* does not use multithreading although this might enhance performance on modern CPU architectures. *ioquake3* utilizes threads through the Simple DirectMedia Library (SDL) 1.2 (SDL_thread), and the platform independent *OpenAL* library for sound. The engine also supports some video playback, but this is also true for the unmodified version.

OpenArena's gameplay is as its predecessor, *Quake III Arena*, primarily focused on the online multiplayer experience. It also supports a simple single player game mode, which behaves as a multiplayer game where the opponents are artificial intelligence controlled clients that simulate players, and the local machine runs the server. The bots are programmed in C, using a well-structured collection of functions for in-game artificial intelligence. *OpenArena* natively supports up to 12 simultaneous players in one match. Single player progress, achievements and settings are saved on the local hard drive. Multiplayer saves primarily the settings, such as user name and character. None are backed up through a cloud solution.

The code quality is in general well-structured and documented, through naming conventions, comments and file directory structure. There is a log system available, and an in-game command window with various options. Technical documentation is found on *OpenArena*'s and *ioquake3*'s official websites and *Wikimedia* based community sites.

The game compiles and runs on Windows, OSX and most Linux based platforms and it thus qualifies as portable. It is one of the enhancements made by the *ioquake3* project, replacing platform specific libraries with cross-platform libraries. Both the *idTech 3* and *ioquake3* engines are primarily first-person shooter (FPS) engines, which somewhat limits the engines' adaptability. It is possible to modify it to support other game genres, but probably not without changing engine specific features.

PlaneShift (Crystal Space)

PlaneShift is an open source community developed multiplayer online role-playing game (RPG) using the open source *Crystal Space* engine. All code and the engine is released under the *GNU General Public License* (GPL), while the data content files are copyrighted to *Atomic Blue Non Profit Corporation* and distributed under the *PlaneShift Content License*. The license details can be found on their official website [17]. There is only one game mode, multiplayer, where you are able to interact and fight with other players or non-player characters (NPC). Some NPCs are controlled by artificial intelligence and can be considered bots, but none possess the AI needed by a virtual player. None of the NPCs are capable of playing the game as a real player. Role-playing is encouraged by the developers, and is described as a highly important aspect of the game. They have created several game rules to encourage roleplaying. Such as the rule stating that you are not allowed to give your character a name associating elements from the modern day real world. All of these rules can however be avoided by setting up your own private server. Each player needs an individual game account, stored in the game's database, together with character progress and other game data. This gives a "cloud like" storage solution with a globally available database, where you can access your game data from any computer. The official database is however currently only one server. Your data may be erased at any given time, because the game is still currently under development with most of the key functionality in place [16].

The game can therefore not be categorized as completely stable, or optimized. Unexpected behavior and bugs do exist as we experienced while testing it, but it never crashed. It is of surprisingly high quality, and will be an interesting candidate when finally released as stable.

Crystal Space, the game's engine, is a highly portable open source software development kit (SDK) for developing 3D applications. It is released under the *GNU Lesser General Public License* and is developed using C++. Various games have been created with the SDK, and it can be classified as highly adaptable. The SDK includes all functionality required by a game, including physics, collision detection, graphics, user input, GUI, and more. It is also possible to find or develop plugins for special needs

[68].

The game itself is a more specialized piece of software, serving its purpose as a multiplayer RPG. It is probably easier to adapt than the previously evaluated games, due to the nature of the engine and the development tools included to make content development easier. PlaneShift has a menu system, a 2D GUI system, some physics, and supports joystick as an alternate input device.

Other Games

The other games were rejected for various reasons. Most because they are too old, including graphical rendering techniques which are obsolete for the project to represent a modern, near state-of-the-art video game.

Vega Strike, an Action Spaceship Simulator, was evaluated but rejected as it turned out to be an incomplete game without any official support for online multiplayer gameplay. The game's graphics is also primarily 2D textures with few 3D models, and is thus not considered a game representative of commercial standards. Other considered, but unfinished open source games are *0 A.D.*, *Ancient Beast*, *Chaotic Rage*, *Dungeon Crawl Soup*, *Flare*, *FreeCol*, *FreeOrion*, *Hedgewars*, *Minetest*, *SpaceZero*, *Teeworlds*, *X-moto*, *SuperTuxKart*, *Sintel The Game*, *Rigs of Rods*, *Unknown Horizons*, and *Unvanquished*.

Other quite popular games such as *BZFlag* ([48]) a *Tank FPS*, and *QuakeWorld* (Cordeiro et al. (2007)[15]) have been rejected even though they have been previously used for computer science. This is primarily because the games date back to the late 90ies [75] and are not good enough, even though they are recently updated.

This also includes games as *The Battle for Wesnoth*, *Battle City*, *Flight Gear*, *FreeCiv*, *Open Hexagram*, *PokerTH*, *StepMania*, *Chocolate Doom*, *Tales of Maj'eyal*, *Advanced Strategic Command*, *Angband*, *Biniax*, *OpenRA*, *Zero-K*, *Crossfire*, *M.A.X.R.*, *TripleA*, *UFO: Alien Invasion*, *Widelands*, *C-Dogs*, *Katana Shoujo*, *Mari0*, *CorsixTH*, *Exult*, *Freesynd*, *Gigolomania* and *NX Engine*.

Other games were also rejected, as they were too similar, if not on the same engine, as earlier selected and evaluated games. This includes games as *Oolite*, *Red Eclipse*, *Doom 3 BFG Edition*, *Xonotic*, *Cube 2: Sauerbraten*, *Smokin' Guns* and *Warsow*.

4.1.4 Selecting a Game

The evaluation indicates that the best video game candidates for research are *Doom 3*, *OpenArena*, and *PlaneShift*. This research was published and presented in Aachen 2013 at the first workshop on *Large Scale Distributed Virtual Environments on Clouds and P2P* (LSDVE), in conjunction with *Euro-Par 2013* [29].

Raaen et al's [50] publication describes an ideal game for implementation of their *LEARS* architecture as a latency in-tolerant high pace action game. Claypool & Claypool (2006)[11] have measured players' latency threshold in various games, and concludes that role-playing games, such as *PlaneShift* have a latency threshold that is 5 times larger than first person shooter games (FPS). It narrows the potential candidates down to the FPS games: *Doom 3* and *OpenArena*. *Doom 3* thus seems like the best choice as it the closest candidate to a modern industry standard game, especially with the recently re-mastered version; the BFG edition [19]. Unfortunately, it does not have support for AI controlled bots for the online multiplayer section of the game. A solution might be to port the single player bots into multiplayer, but the bots won't probably work very well as they aren't natively aware of the multiplayer game rules. *OpenArena* is therefore the only remaining candidate for implementation of virtual clients.

OpenArena

OA has a quite active community and it took some weeks to get acquainted with the source code and the development community. OA is primarily a content focused development project and most of their updated documentation revolves around content and not programming. The developers have hence primarily worked on developing game content and logic, which is partly defined in the *Quake Virtual Machine* (QVM) files that are utilized by the engine. Especially confusing is the fact that they have chosen to separate the development of the QMVs and the engine, which are no longer fully compatible with each other. It required significant amounts of time and some forum posts to understand fully. I made multiple unsuccessful attempts of merging the engine source with the QVM source, something my task required as I needed to have a codebase with direct support for communication between the components of the game. It turned out that there are so many undocumented differences that even the veteran OA developers discouraged the merge [71].

My efforts did however inspire the coders of the community, so the OA engine was shortly after merged together with the latest version of the official codebase of *ioquake3* [26], and their QVM remake (OAX) codebase was for the first time added to *git* version control system on *GitHub* [53]. I contributed with testing, primarily on the engine, and several bugs were discovered in *ioquake3* [65], *MinGW* [27], and even in *Simple DirectMedia Layer* (SDL) 1.2.15 [28]. Some of the proposed resolutions were patched into the engine. OA's codebase is nevertheless too cluttered for a smooth implementation process of virtual clients.

Quake III Arena (ioquake3)

The *ioquake3* project has in contrast to OA always utilized version control and kept the game logic (QVMs) and the engine within the same codebase. This makes engine and QVM development an easier and cleaner procedure, which is the reason I chose to develop the implementation in Q3A rather than OA. The source code is open source, but the Q3A data files needs to be legally obtained through purchase. I chose *ioquake3* over the original *idTech 3* codebase as I consider *ioquake3*'s modifications positive enhancements.

Project	Type	Description
ioquake3	Executeable	Full Game (GUI)
ioq3ded	Executeable	Dedicated Server (Terminal)
cgame	Dynamic Library/QVM	Game Client Logic
game	Dynamic Library/QVM	Game Server Logic
ui	Dynamic Library/QVM	Graphical User Interface
renderer_opengl1	Dynamic Library	OpenGL 1 Renderer
renderer_opengl2	Dynamic Library	OpenGL 2 Renderer

Table 4.3: Game Components of the ioquake3 Engine

4.2 Architecture of Quake III Arena

My previous research on open source games useable for computer science concluded that the video game and open source project *OpenArena* is one of the best available candidates as of today. It is a game with custom community developed game data[2] and a slightly modified version of the *ioquake3* engine. It is a currently maintained and enhanced version of *id Software's id Tech 3* engine - the original game engine behind *Quake III Arena*, released as open source in 2005. *Quake III Arena* is thus just as applicable as *OpenArena*, but it requires you to purchase a copy of the game in order to retrieve the game data. Both are first person shooter games with support for both single- and multiplayer game modes.

Both the original Q3A and the modified *ioquake3* game engine are highly modular pieces of software. *John D. Carmack*, the programmer of the original engine, explained that he decided to create a custom virtual machine (VM) and VM language for portability and better security than traditional dynamic libraries [9]. The game consists of a base executable, Quake Virtual Machine objects (QVMs), and third-party libraries. The base executable is a framework utilized by the QVMs for handling I/O operations, including user input, rendering, network, file management, audio playback, and cross-QVM communications. The QVMs themselves define the program flow and game logic. The engine's QVM system also supports loading and running DLLs instead of their own *.qvm* format, for easier debugging and development [30].

4.2.1 ioquake3 Components

The primary components of the *ioquake3* engine consist of the components listed in table 4.3. Section 6.1 explains how you can obtain, build and run the game and the virtual clients.

[2]Game data as in game assets including binary files such as textures, models, sounds, etc., and other text based assets.

4.2.2 Client/Server Communications

Network communications in Q3A have been implemented in a quite elegant manner, using a logical OR based *snapshot system* that compensates for lost snapshots (*UDP*), and ensures that only the necessary data is transmitted through the network. The engine uses a module called *netchan* for low-level packet operations, including encryption and compression. Encryption is done using a pre-shared key that is established between the client and server on client connection. Compression is done using a static Huffman table [21].

The client to server communication is quite straightforward; the client sends a user command every time the player performs an action, changing the player state. The server is more complex and sends snapshots to each of the connected client at a fixed rate. The transmitted snapshot is unique for each client and contains the delta state of each entity in the game that has changed since the last transmitted snapshot, including the client's player state. The snapshots are limited to transmit only the game entities within the view area of the player. Both the client and server have implemented latency compensation through extrapolation and interpolation, but the client knows that the server is the master and adjusts its entities from the snapshots [30].

Chapter 5

Implementating Virtual Clients

This chapter documents the technical implementation of virtual clients in *Quake III Arena*. The first section introduces the initial concepts of the implementation, and specific implementation details are presented in the following sub-sections. Code examples are provided together with detailed explanations that discuss the various design decisions.

5.1 Implementing Virtual Clients

The Q3A bot library was developed by J.M.P. van Waveren as his Master's thesis in 2001 at the Delft University of Technology [73]. The bots use a 4-layered architecture where the two last layers are implemented in the server logic's VM while the two former are implemented as a standalone bot library included in the engine. He decided to hard-code the two of the layers into the server's game logic for simplicity as the bots depend heavily on game logic. This makes the entire bot library more static, thus harder to reuse and port.

Virtual clients do need AI, so our initial plan was to port the bot library from the server's game logic to the client logic. The hard-coded layers of the bot library make this a complex and time-demanding task. It requires migrating both the bot library and server-specific dependencies into the client code. Alternatives are modifying the server logic to behave as a client, or to implement a *proxy*. We implemented the latter solution as the former has similar drawbacks as the original plan.

We were inspired by Q3A's implementation of single player and local machine game modes. The design solves a similar problem to ours because it requires some server-side logic within the client in order to function properly. This has been solved without implementing any server code within the client. Single-player mode is launched in two steps. It starts by launching a local and "hidden" server with a specified map. Once done, it launches the client logic, which connects to the hidden local server. The virtual client implementation is more or less an extension of this procedure. It connects to a real server and retrieves its game specific data (map, entities, etc.). This is used to create a hidden local server with the same game data as the real server; a shadow copy. A bot is added to the shadow server, and the engine bridges the communication between the bot and the real server, working like a proxy. The entire implementation is defined by the following procedures:

1. Configuration through cvars
2. Shadow server & bot initialization
3. Synchronization & proxy communication

5.1.1 Configuration Through Cvars

The virtual client implementation is designed to run through the console, and not the in-game GUI. It is both faster and easier to implement than a graphical menu, and enables virtual clients to be launched from shell scripts. I decided to use the engine's built-in *console variable* (cvar) configuration system to trigger the virtual client functionality from the console. The engine is built to accept commands through program parameters on launch, enabling cvars to be set through parameters on program initialization. A virtual client can be hence be enabled by setting the proper cvars as program arguments, followed by connecting to a dedicated server. Cvars may also be set through engine configuration files. The following example illustrates how to launch the engine with the *virtualClient* cvar set to *1*:

```
./ioquake3.x86 +set virtualClient 1
```

Other implemented cvars are *virtualClientSkill*, *virtualClientBot*, and *virtualClientName* which are described in section 6.1. Every cvar affects the virtual client differently and comes with a default value if not set as a parameter or in a configuration file. They are stored in the engine as *cvar_t* structures, which are linked with the cvar system on initialization. The following example is fetched the cvar declarations in *./code/qcommon/common.c*. It links *virtualClientSkill* with the cvar system, provides a default value if not already set, and validates its value:

```
// Retrieve bot skill (worst 1-5 best)
com_virtualClientSkill =
     Cvar_Get("virtualClientSkill", "4", CVAR_LATCH);
Cvar_CheckRange(com_virtualClientSkill, 1, 5, qtrue);
```

The cvar system stores each cvar as integer, float, and string (char array) in their representative *cvar_t* structures. It enables support for all the three datatypes and prevents the need for computationally heavy conversions. The following example illustrates how the engine can use the *com_virtualClient* cvar to detect the support for virtual clients.

```
if (com_virtualClient->integer) {
  // Virtual client specific code here

  if (com_virtualClient->integer < 2) {
    // GUI specific virtual client code here
  }
}
```

5.1.2 Shadow Server & Bot Initialization

Some conditions need to be met in order to launch the shadow server: The *virtualClient* cvar has to be set, the client must be connected to a real game server, and the client related data structures must contain valid data. It could have been done by implementing multiple special-case conditional checks in the engine's primary game loop, but it would break with the engine's high-cohesion architecture and require that the conditionals are processed every single frame.

A better approach is to implement it in a rarely called function that also requires some of the same conditions to be met. It was thus decided to place it in the client's game-state parser function, *CL_ParseGamestate()* in *./code/client/cl_parse.c*, which is called every time the real server transmits a game-state update to the client. Game-state update does only concern major modifications to the active game on the server, such as a map-switch or a new game mode. It is thus only transmitted to a client at its initial connect and when the server needs to reload the game. This implies that the *CL_ParseGamestate()* is a rarely utilized function that is only called if the client is connected to a server. The original functionality in the function also ensures that the required client data structures contain valid data, providing ideal conditions for launching the shadow server. A stripped down version of the function containing the shadow server launch procedure is provided below.

```
void CL_ParseGamestate( msg_t *msg ) {
  /* Game state parsing */

  // Launch the shadow-copy of the real server if required
  if (com_virtualClient->integer && !com_sv_running->integer) {
    char *serverInfo;

    // Retrieve server info on the real server
    serverInfo = cl.gameState.stringData
      + cl.gameState.stringOffsets[CS_SERVERINFO];

    // Set up local server params with data from the real
    //  server to mirror it locally
    Cvar_Set("fraglimit",
      Info_ValueForKey(serverInfo, "fraglimit"));
    Cvar_Set("timelimit",
      Info_ValueForKey(serverInfo, "timelimit"));
    Cvar_Set("g_gametype",
      Info_ValueForKey(serverInfo, "g_gametype"));
    Cvar_Set("sv_maxclients",
      Info_ValueForKey(serverInfo, "sv_maxclients"));
    Cvar_Set("g_maxGameClients",
      Info_ValueForKey(serverInfo, "g_maxGameClients"));
    Cvar_Set("capturelimit",
      Info_ValueForKey(serverInfo, "capturelimit"));

    // Start the local virtual server, utilizing
    // the engine's command system
    Cbuf_ExecuteText(EXEC_NOW,
      va("devmap %s\n",
            Info_ValueForKey(serverInfo, "mapname")));
    }
}
```

The function verifies that the *virtualClient* cvar is set and that the local shadow-server is not already running. It then retrieves the real server's gamestate data from the client's *cl* structure, and extracts its values to create the proper configuration for the shadow server through cvars. The remaining code executes a *devmap* command through the engine's own command system, with the real server's map name as parameter. This triggers the engine to call the same procedure that is called when launching a hidden single player game server, which has been slightly modified with virtual client conditionals.

The next step is to add the bot to the shadow server, something that should only be done on the first received snapshot from the real server. It has been done by using the same approach as with the shadow server, modifying an existing function, *CL_ParseSnapshot()*, in *./code/client/cl_parse.c*. The function guarantees that the client is connected and that the shadow server is running, as a game state is always handled before the first snapshot arrives. Its existing functionality already contains a procedure of identifying the very first snapshot. It is utilized to perform a call to a new function, *SV_CreateVirtualPlayer()*, with various data from the initial snapshot as parameters. The snapshot data contains the real server's game time, the state of the game's entities, player data, and an index into the entity array. It is used to create the bot in the local shadow server, and to synchronize the shadow server with the real server. This procedure is provided below.

```
void SV_CreateVirtualPlayer( int serverTime,
    int parseEntitiesNum, int numEntities,
    entityState_t* entities, playerState_t* ps )
{
  int         i;
  client_t    *cl;

  if (!com_sv_running->integer || !com_virtualClient->integer
      || virtualClientInitialized) {
    return;
  }

  // Store the time of virtual client creation
  virtualClientInitialized = ps->commandTime;

  // To ensure that the add command selects the proper client
      slot
  for (i = 0, cl = svs.clients; i < ps->clientNum; i++, cl++) {
    if (cl->state == CS_FREE) {
      cl->state = CS_VC_OCCUPIED;
    }
  }

  // Concatenate the arguments for the QVM syscall
  Cmd_TokenizeString(va("%s %d %s %s %p", com_virtualClientBot
      ->string,
    com_virtualClientSkill->integer, "0", com_virtualClientName
        ->string, ps));
  Cmd_Args_Sanitize();

  // Synchronize time with the real server
  Sys_Sleep(svs.time);

  // Add the bot to the local shadow server
  VM_Call(gvm, GAME_ADD_VIRTUALCLIENT, ps);

  // Concatenate new arguments for the next QVM syscall
  Cmd_TokenizeString(va("%p %p", entities, ps));
  Cmd_Args_Sanitize();

  // Update the shadow server's entities with real server data
  VM_Call(gvm, GAME_UPDATE_VIRTUALCLIENT, parseEntitiesNum,
      numEntities);

  // Re-enable the client slots
  for (i = 0, cl = svs.clients; i < ps->clientNum; i++, cl++) {
    if (cl->state == CS_VC_OCCUPIED) {
      cl->state = CS_FREE;
    }
  }
}
```

The function starts by ensuring that the shadow server is running, that virtual client support is enabled, and that the function has not been called previously. The latter uses a variable that is immediately set afterwards with the time of the bot's creation. The following loop iterates through the pre-allocated server array of client slots and sets every client slot before the provided player's ID number to occupied. It is required to ensure that the following system call to the server QVM selects the same client slot for the player as on the real server. Entity updates would become difficult otherwise. The function then concatenates the parameters to the server QVM call as a string, and adds it to the engine command system's argument stack. It is required as the QVM call procedure, *VM_Call()*, only support integer parameters. Pointers could theoretically have been casted to integers and used directly, but it would not work in 64-bit builds of the game because integer and pointer size differs on the 64-bit architecture. It then sleeps some milliseconds to synchronize the shadow server's game time with the real server. Development tests demonstrated that this was necessary to prevent the bot from performing pre-mature in-game actions. It then calls the server QVM system call to add the bot, which leads to a QVM function call in *./code/game/* that is shown below. The remaining functionality of the procedure performs a new server QVM call to synchronize the entities, and re-enables the occupied slots. The entity update QVM call is documented in 5.1.3 on the following page.

```
static void addVirtualClient() {
  char botName[21];
  char skillStr[4];
  float skill;
  char team[5];
```

```
    char vcName[21];
    char ps[15];
    playerState_t* psp = NULL;

    // Retrieve botname and skill from argument stack
    trap_Argv(0, botName, sizeof(botName));
    trap_Argv(1, skillStr, sizeof(float));

    // Verify and cast the skill
    if (!skillStr[0]) {
      skill = 4.f;
    } else {
      skill = (float)atoi(skillStr);
    }

    // Retrieve team, client name, and player state pointer
    // from argument stack
    trap_Argv(2, team, sizeof(team));
    trap_Argv(3, vcName, sizeof(vcName));
    trap_Argv(4, ps, sizeof(ps));

    // Cast the pointer string to a real pointer
    sscanf(ps, "%p", &psp);

    // Add the bot using a slightly modified version of the QVM's
    // native G_AddBot function
    G_AddBot(botName, skill, team, 0, vcName, psp);
}
```

The function retrieves necessary data from the engine command system's arguments stack, which all are stored as strings. It casts and verifies the arguments to proper values and performs a call to the slightly modified version of the QVM's function for adding bots. The modifications primary concerns the support for providing a pre-defined player state. It makes the function position the bot at the exact same location as on the real server and synchronizes the player state data. The local running engine has now a complete copy of the real server with all entities synchronized with the latest snapshot of the real server.

Other smaller modifications have also been made in various files to ensure proper virtual client functionality. Common for them all is that they all check the *virtualClient* cvar to ensure that the modifications do not apply if it is not enabled.

5.1.3 Synchronization & Proxy Communication

A local shadow server with a bot initially positioned at the same location as the client on the real server, does not do much good. The bot library immediately starts processing the bot's AI, leading to player actions that only occur on the shadow server. These actions need to be captured and transmitted to the real server. This is implemented by exploiting the engine's event queue system. All engine input, including bot actions, are placed in an event queue. The various components pulls and processes events from this queue that are within their individual responsibilities [57]. Bot action forwarding can thus be implemented by copying the bot's commands (actions) to a client command and place it in the event queue. The client system will then automatically capture the client command and transmit it to the real server, just as if it was the client's direct command. This is implemented with the three first lines of code in the server function presented below, located in *./code/server/sv_client.c*:

```c
void SV_ClientThink( client_t *cl, usercmd_t *cmd ) {
  if (com_virtualClient->integer) {
    CL_AddUserCommand(cmd);
  }

  cl->lastUsercmd = *cmd;

  if (cl->state != CS_ACTIVE) {
    return; // may have been kicked during the last usercmd
  }

  VM_Call(gvm, GAME_CLIENT_THINK, cl - svs.clients);
}
```

The server function is modified to add the received user command to the client's command queue if it is running as a shadow server (virtual client enabled). It uses the *CL_AddUserCommand()* function to enqueue the command into the client's queue, which is implemented in the client code. Calling a client function from the server code is not good in terms of engine architecture, because it makes the server component dependent of a specific client implementation. A better solution would be to utilize an abstract inter-communication procedure between the engine components, as discussed in section 2.2.2. The engine does however not provide this, and it is notable that several other server functions utilize client calls in the unmodified engine source. It was thus decided to keep the presented solution for simplicity.

Another client function is altered to block keyboard and mouse actions that normally produce client commands. Other input is still allowed, enabling the user to access the in-game console and game menu. This applies only when running a virtual client with a GUI, as it is completely ignored when running from the console.

This functionality is alone enough to have a virtual client that interacts relatively well with its surrounding environment, but it is not a realistic client. It will fail to interact with other players and never generate combat traffic. Updates from the real server need to be transmitted to the shadow server in order to synchronize all entities, including other players and the bot itself. The latter takes account for the latency between the shadow and the real servers, and makes the bot adjust itself accordingly. Updates are transmitted from the server at a given interval, with all data stored in snapshots.

Forwarding snapshots from the real dedicated server to the local virtual server is a bit more cumbersome than forwarding bot commands, and introduces the more unstable part of the implementation. The real server transmits updates to the client once every frame, but the updates are limited to only include the modified entities within the client's view area [21]. Also, the actual game times on the two servers are never the same. This is solved by copying the player and entity states from each snapshot received from the real server into the bot and entity structures of the virtual local server. This happens in the end of the client frame function, *CL_Frame* in *./code/client/cl_main.c*, which ensures that all snapshot entities have been processed before transmitting them to the shadow server. A server function, *SV_UpdateVirtualServer*, is called if the shadow server is running and is provided with extracted snapshot data as parameters. The server function is implemented in *./code/server/sv_main.c* and contains the following:

```
void SV_UpdateVirtualServer( int serverTime,
    int parseEntitiesNum, int numEntities,
    entityState_t *entities, playerState_t* ps)
{
  if (!com_sv_running->integer || !com_virtualClient->integer
    || !virtualClientInitialized
    || ps->commandTime == virtualClientInitialized)
  {
    return;
  }

  // Add the arguments for the QVM syscall
  Cmd_TokenizeString(va("%p %p", entities, ps));
  Cmd_Args_Sanitize();

  VM_Call(gvm, GAME_UPDATE_VIRTUALCLIENT, parseEntitiesNum,
      numEntities);
}
```

The function starts by ensuring that all demands are met in order for it to continue, before it pushes the player state and entity pointers to the engine command component's argument stack. It completes by performing a QVM system call to the server QVM that performs the updates on its entities. The QVM function that performs the actual updates is located in *./code/game/ai_main.c* and contains the following:

```
void BotUpdateVirtualClient( int parseEntitiesNum,
  int numEntities )
{
  int i;
  char psBuff[15];
  char entsBuff[15];

  // Retrieve entity and player state pointers from arg. stack
  gclient_t *cl = NULL;
  playerState_t *ps = NULL;
  entityState_t *ents = NULL;

  trap_Argv(0, entsBuff, sizeof(entsBuff));
  trap_Argv(1, psBuff, sizeof(psBuff));

  // Convert the strings to actual pointers
  sscanf(entsBuff, "%p", &ents);
  sscanf(psBuff, "%p", &ps);
```

```

// Ensure that we have valid entities
if (!ents) {
  return;
}

// Loop through all provided entities
for (i = 0; i < numEntities; ++i) {
  // Fetch entity data. 8192-1 == MAX_PARSE_ENTITIES.
  entityState_t* ent =
    &ents[(parseEntitiesNum + i) & (8192 - 1)];
  gentity_t* gent = &g_entities[ent->number];

  // Copy in the new entity state
  Com_Memcpy(&gent->s, ent, sizeof(entityState_t));

  // Special case handling for players
  if (ent->eType == ET_PLAYER) {
    // If not previously initialized on local shadow server
    if (!Q_stricmp(gent->classname, "clientslot")) {
      // Need to create a local playerstate as ClientBegin
      // zeroes out the client->ps
      playerState_t playerState;

      // Set inital state, will be updated next frame.
      Com_Memset(&playerState, 0, sizeof(playerState_t));
      VectorCopy(ent->pos.trBase, playerState.origin);
      VectorCopy(ent->apos.trBase, playerState.viewangles);
      playerState.stats[STAT_HEALTH] = 125;

      // Spawn the entity on the server
      ClientBegin(ent->clientNum, &playerState);
    } else {
      // Player already registered. Check if dead.
      if (ent->eFlags & EF_DEAD) {
        if (gent->client)
          gent->client->ps.pm_type = PM_DEAD;
      } else {
        // Needed to reset dead flag
        gent->client->ps.pm_type = PM_NORMAL;

        // Update the player's position
        gent->r.currentOrigin[0] = ent->pos.trBase[0];
        gent->r.currentOrigin[1] = ent->pos.trBase[1];
        gent->r.currentOrigin[2] = ent->pos.trBase[2];

        // Update the player's angles
        gent->r.currentAngles[0] = ent->apos.trBase[0];
        gent->r.currentAngles[1] = ent->apos.trBase[1];
        gent->r.currentAngles[2] = ent->apos.trBase[2];
      }
    }
  }
}

// Ensure we have a proper player state
if (!ps) {
  return;
}
```

```

    // Retrieve the virtual client
    cl = g_entities[ps->clientNum].client;

    // Update its player state
    if (cl) {
      Com_Memcpy( &cl->ps, ps, sizeof(playerState_t) );
    }
  }
```

The function loops through all the provided entities from the snapshot and updates their entity state data structures. Some special handling is required for client entities (players) as the bot needs to know whether they are enemies, alive or on the map at all (spectators). It also updates the bot's player state that ensures a complete synchronization with the real server. It works well and the virtual client is completely able to interact properly with other clients in the game. The same goes for static entities, as it now knows whether another player has claimed an item on the map.

5.2 Summary

Virtual client functionality has been implemented by running a bot on a local replica of the real server, where client commands and server updates are communicated through the kernel by exploiting engine features. The kernel works like a proxy that transfers data between the real and shadow server that does not know about each other. A visual representation of the entire procedure is illustrated through an UML sequence diagram in figure 5.1. The following paragraph explains the figure in detail.

It all starts with a *handshake* where the client requests a connection to the server by providing a randomly generated *challenge*. The server verifies that the client's IP addressed is not blocked and that no other connected client has the same challenge. An accept message is returned to the client if everything is good. The challenge is now used to encrypt the data transmitted between the client and server. The client transmits a connect message containing player data. The server accepts the message, spawns the client, and returns a *baseline* to the client, which is a clean snapshot containing all data on all entities. This triggers the function that launches the local shadow server based on the received game data. Some milliseconds pass by before the client receives its first snapshot that triggers the creation of the bot. The client sleeps to synchronize its game time with the real server and performs the first entity synchronization. Everything is now properly initialized and the game loop ensures that bot commands and server updates are continuously transmitted between the real and shadow server.

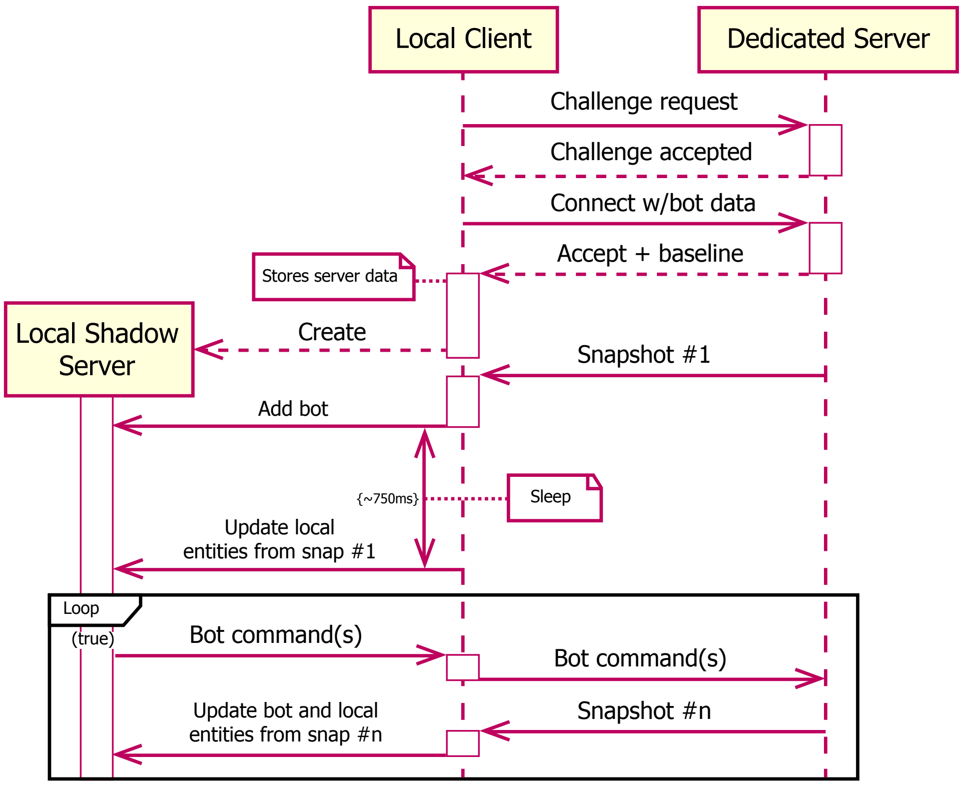


Figure 5.1: UML Sequence Diagram of the entire Virtual Client Procedure

Chapter 6

Installation & Usage

This chapter provides instructions for installation and usage of virtual clients in *Quake III Arena*. The first section describes how to install and build the game on your local system, while the second explains how to use the virtual clients. Some shell scripts have been developed for easier usage and are provided together with the source. These are documented with examples of usage. The installation and usage instructions may also apply to both *OpenArena* and the unmodified original game, and can be useful for anyone interested in those games.

6.1 Installation

This section explains how to install and build the virtual client-enabled game. Instructions are provided on how to obtain the game content files and build alternatives are presented.

6.1.1 Installing the Binary Data Files

Only the source code of *Q3A* is released as open source under the GNU General Public License version 2. The binary data files, including 3D models, textures, audio files, etc. are not open and needs to be obtained from a original CD or by purchasing the game online [20]. Use an installer or manually extract the binary files to an appropriate location on your system. The *ioquake3* project's official website provides installers that includes the latest patches and creates the proper directories and files. All you have to do is to extract the primary binary data file *baseq3/pak0.pk3* and the product key file *baseq3/q3key* from the purchased product and place them in the *baseq3* sub-directory of the installed *ioquake3* directory [49]. Test your install by running the *ioquake3* executable.

6.1.2 Building ioquake3 with Virtual Clients

The source code of the *ioquake3* virtual client implementation can be obtained from my *GitHub* git repository [13]. The *master* branch is nothing more than a copy of the original *ioquake3* project's official Git branch [14].

My implementation is found in the *ProxyClient* branch. The engine and QVMs can be built using *Make* (*nix) or the *Visual Studio 2013* project located in *./misc/msvc12/ioq3.sln* (Windows). Detailed install instructions and documented build alternatives can be found by investigating the *Makefile* or in the official *README* [14].

Retrieving and building the engine using Make:

```
git clone git@github.com:stigmh/ioq3.git
cd ioq3
git checkout ProxyClient
make
```

The first instruction above clones the source code of the external git repository into subdirectory named *ioq3* on your local drive. The next navigates into the subdirectory, which is now a local git repository. The remaining instructions jump into the proper git *branch* and start the automated build process using *Make*. Resulting build binaries can be found in a sub-directory of the *./build* directory on successful compilation. Which exact sub-directory depends on your platform and build preferences. Test the build by running the executable with the *+set fs_basepath* argument, which tells the executable where to locate the data files. Note that it requires *Simple DirectMedia Layer* (SDL) 1.2.15 to be installed on your system. SDL resources and guides can be found at their official website [37].

Example of running compiled executable on OSX:

```
./ioquake3.x86 +set fs_basepath /Applications/ioquake3/
```

Be aware that this approach will only use the compiled *ioquake3* executable, and not the compiled QVMs. You will have to do some file copy operations and provide some additional arguments in order to use the compiled (development) QVMs. First, you need to either copy the compiled dynamic libraries (*.dll/*.so) into the *baseq3* directory of the Q3A install directory. Alternatively, create two directories (e.g.: *virtualclient/vm*) in the Q3A install directory and copy the compiled **.qvm* files there. Second, run the executable with arguments instructing it to use your compiled QVMs rather than the ones located in the *pk3* data files.

Example of running the executable with custom built dynamic library QVMs:

```
./ioquake3.x86 +set fs_basepath /Applications/ioquake3/ +set
    sv_pure 0 +set vm_ui 0 +set vm_game 0 +set vm_cgame 0
```

6.2 Usage

This section explains how to manually enable virtual client support in the engine, and how to use it for server and network load generation. Several shell scripts are described that can be used for an automated and easier procedure.

Cvar name	Description
virtualClient	Integer, 1-2. Whether to launch as a virtual client. 0: no. 1: yes with GUI. 2: yes with terminal.
virtualClientSkill	Integer, 1-5. 1 is worst, while 5 is best.
virtualClientBot	String. Which bot script file to utilize, also used for model.
virtualClientName	String. Name of client on server, can be anything.

Table 6.1: Description of implemented cvars to control the virtual clients.

6.2.1 Running ioquake3 with Virtual Clients

Four configurable cvar variables have been added to the engine, which are used to dynamically control the virtual clients. The engine's cvar system enables them to be set and modified through program arguments, the in-game console, or through an engine configuration file. Table 6.1 lists the implemented cvars with descriptions. The most essential cvar is *virtualClient* that defines whether or not to enable virtual client support. It defaults to zero, which makes it ignore all modifications and run like the original game. Setting it to *1* launches the game with the rendering system enabled and a virtual client that you can spectate from its point of view, which can be handy for development. The user gets full access to the engine's graphical user interface (GUI). Setting it to *2* launches the game in console mode with a virtual client. Launching it as console requires much less resources than with GUI. Running in console does not need to perform graphics processing, nor load the menu QVM and graphical elements. It is useful when multiple virtual clients are to run on the same machine. Note that it is advised to always connect to a server on the launch of a virtual client, through the *+connect* parameter.

Running a virtual client with GUI enabled and a custom name:

```
./ioquake3.x86 +set fs_basepath /Applications/ioquake3/ +set
    sv_pure 0 +set vm_ui 0 +set vm_game 0 +set vm_cgame 0 +set
    virtualClient 1 +set virtualClientName Stigmha +connect
    127.0.0.1
```

6.2.2 Shell Scripts for Easy Usage

The git codebase [13] provides some shell script files to make development and launching virtual clients easier. They are located in two subdirectories of *./misc/virtual_client/*, one directory for Windows (*.bat) and one for Unix based systems (*.sh). All depends on a local file that you need to create, called *baseq3path.local.(sh/bat)*, which you can find and example of in the README files. Script documentation and examples are available within two *README* files and the comments of the script files. They also require that you have installed the game and built the virtual client binaries as described in 6.1 on page 61. This section documents the scripts and their usage. All are configured to run with the development QVMs as described in the previous section.

install_(so/dll)

This script should be run after building the engine or any of the QVMs. It copies all the dynamic libraries (*.so/.dll) to the proper location of the install directory, ensuring that you are working on your compiled version of the QVMs instead of the natives located in the *pk3* files. The script takes no arguments, but requires that your local *baseq3path.local.(bat/sh)* is configured properly.

launch_virtualclient

Launches a single virtual client. It has five optional arguments in the following order: server, virtual client mode, user name, skill, and bot. The server argument should either be the IP or domain name of the game server; default is 127.0.0.1 (*localhost*). Virtual client mode specifies the *virtualClient* cvar that you can modify to run it in either GUI or console mode, default is GUI (1). The user name specifies the in-game name of the connected client; default is *VirtualClient*. Skill defines the virtual client's (bot) skill level and should be between 1 and 5, default is 4. Bot specifies which model and bot script to use, default is *sarge*. Example of use:

```
./launch_virtualclient.sh nagios.nith.no 1 Stigmha 3 orbb
```

launch_multiple_vcs

Launches multiple virtual clients with random properties in console mode, all launched as background processes with output redirected to *null*. Accepts two arguments: number of clients and server. Server is the same as in *launch_virtualclient*. Number of clients specifies how many virtual client processes to launch, default is 8. Note that the original engine has hard coded support for maximum 64 clients on one server. This can easily altered by changing two C macros in the engine's source. Example of use:

```
./launch__multiple_vcs.sh 32 nagios.nith.no
```

launch_dedicated

Launches a dedicated ioquake3 server with the configuration available in *./misc/virtual_client/dedicated_server.cfg*, which is a server configuration file made especially for the use of virtual clients. It has a huge time and frag limit, ensuring that map switches do not occur. No arguments are available.

connect_localhost

Launches the game and connects to a server on the local machine. Sets the client name to *Stigmha* and the model to *sarge/krusade*. No arguments are available.

benchmark.sh & manual_benchmark.sh

These files are only available in the *Unix* directory. They are scripts to benchmark a *Unix* based game server utilizing *SSH*. Primarily used to measure how the virtual clients affect server performance. Collected data is used for evaluation and to generate some of the figures in this thesis. Both can be configured through parameters and by modifying the variables at the beginning of the files.

Part IV

Results

Chapter 7

Results and Evaluation

7.1 Server Load Generation

A stress-test with 0-48 virtual clients was performed and monitored to evaluate how the implementation loads server hardware and networks. It measured CPU load, memory usage, and network bandwidth consumption on the server. The test could potentially have been done with up to 64 clients, but the server became overloaded with 48 clients and were unable to process any additional connections. It resulted in connection timeout failures for both *Quake III Arena* and *SSH* connection attempts.

The benchmark was performed on an older *Debian 6.0* Linux server running an unmodified 32-bit version of *ioquake3*, with a single core *Intel(R) Xeon(R) X5650* 64-bit 2.67GHz CPU and 250MB RAM. The virtual clients and the benchmarking procedure were launched from a more powerful multi-core machine running *Kali Linux*. Everything was automated through the use of shell scripts which use static time intervals for precision, these are documented in section 6.2.2 on page 63. All measured values are calculated from the average of 10 readings.

Network Traffic Load

Figure 7.1 shows how the virtual clients affect the server's network bandwidth, with values measured in kilobytes per second (Kb/s). The *Data in* graph is close to linear because each client is responsible for transmitting its own updates to the server [21], the bandwidth increases proportionally with the amount of clients. *Data out*'s graph is closer to quadratic as each client needs to receive updates from all the other clients within its respective *view area* [21].

Both graphs show that the virtual clients successfully generate network traffic that affects the server's overall bandwidth load. Each client continuously transmit their player actions to the server, which is processed and leads to snapshot updates that needs to be transmitted to all the connected clients. The more players, the larger the snapshots, and the more snapshots are transmitted every frame.

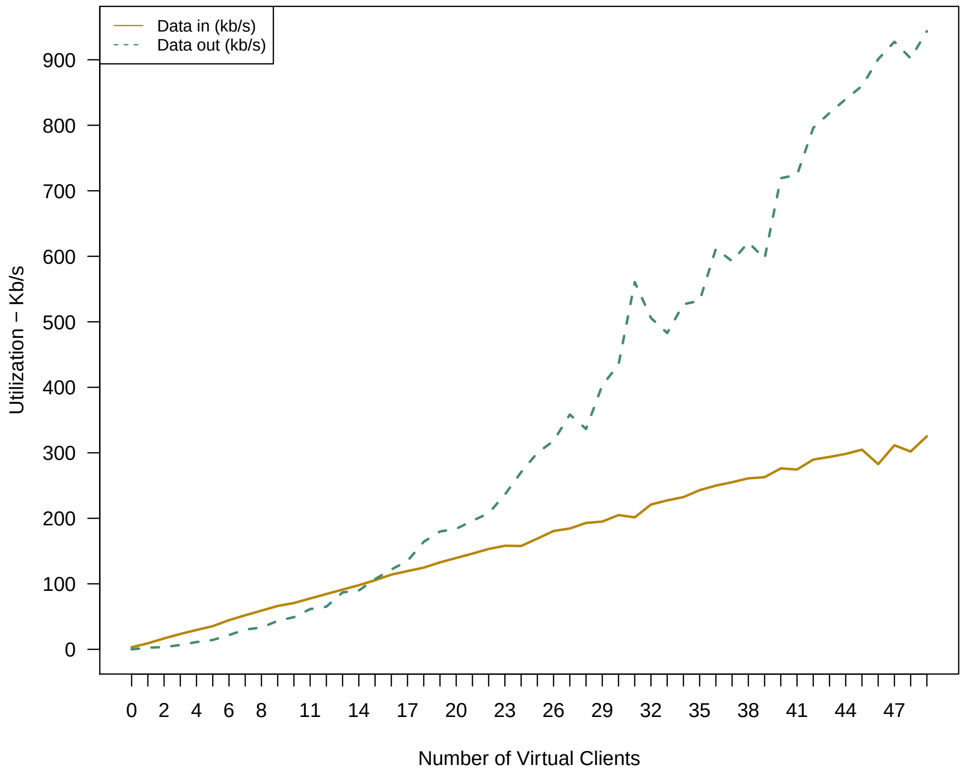


Figure 7.1: Network Load on Server with 0-48 Connected Virtual Clients

CPU & Memory Utilization

Figure 7.2 illustrates the server's CPU and memory utilization and is measured in percentage (%). CPU utilization increases until it stagnates at approximately 71% utilization with 21 connected virtual clients. The server starts to struggle as it becomes overloaded, so the operating system needs to share the available processing capabilities among other processes. A more powerful multi-core server will likely manage to serve more clients before it stagnates. An entire core can be dedicated to the game server and a higher clock frequency will enhance the processing speed. A multi-core machine may also run multiple instances of the server software, allowing an even greater amount of simultaneous connected players.

Memory usage remains constant as the game engine always pre-allocates enough memory for the max number of supported clients (64) on start-up. This allows the server to dynamically manage its clients without performing any expensive dynamic memory operations. The server also pre-allocates memory for dynamic entity management for the same reason. This allows it to create and delete new entities, such as projectiles, without much overhead. Disadvantages are that the engine must be altered and recompiled to support more than 64 players, and that it will always reserves enough memory for its maximums supported clients and entities. Larger portions of the reserved memory may never be utilized. Memory is however considered cheap, so it is worth it in terms of performance. The maximum number of players may be increased by modifying two macros in the source code following by re-building the engine.

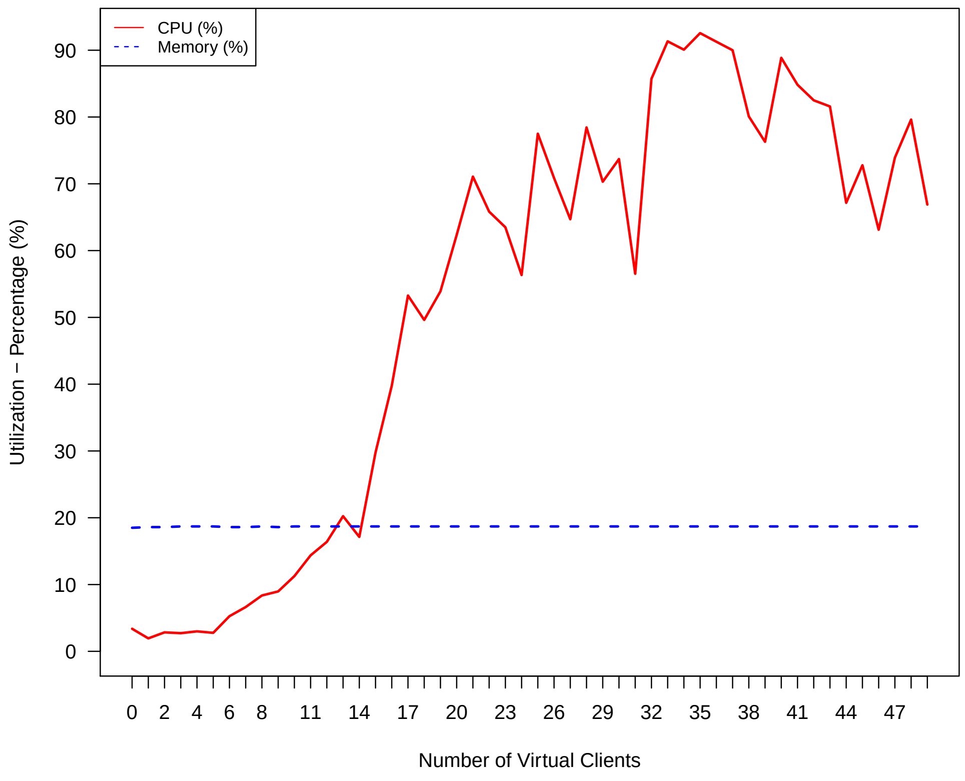


Figure 7.2: Server CPU & Memory Utilization with 0-48 Connected Virtual Clients

Mode	CPU	Memory	Bandwidth in	Bandwidth out
Regular player	252.42%	79 MB	1.64 Kb/s	5.80 Kb/s
VC Graphical	234.26%	94 MB	2.00 Kb/s	5.14 Kb/s
VC Console	6.94%	29 MB	1.41 Kb/s	6.00 Kb/s

Table 7.1: Various Client Types and their System Load

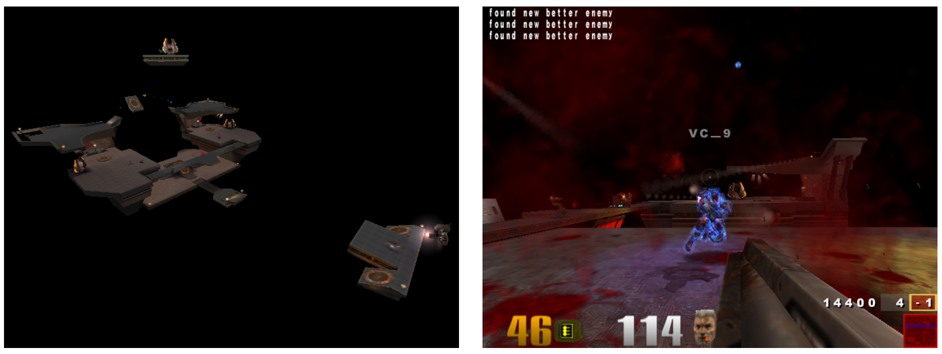
7.2 Virtual Client Resource Usage

Our implementation is capable of running in two different modes; graphical and console. Graphical mode allows the user to observe the virtual client from the bot's point of view, and all the visual components of the game are present. Console mode is based on the dedicated server modifications to the game, and presents the user with nothing but a console that prints the engine's output. All virtual clients where launched in console mode for the server benchmarking.

Table 7.1 shows an average of the various client types' resource usage, all measured with a single client. The data is retrieved from a host with a quad core 2.6Ghz CPU (max 400%) and 4GB memory. The console version is executed in one thread, while the two others utilize six. It shows that the console client requires significant less processing power and memory than the others, while the network differences are minimal. The reason is that the console client does not load and process the audio-visual assets of the game, including the entire UI QVM. Running a GUI enabled virtual client utilizes more resources than a regular player, as it requires additional memory and processing for the local shadow server.

The results in table 7.1 show that this specific host is theoretically capable of running up to 57 virtual client instances simultaneously without getting overloaded. A more powerful host is likely to handle more, but multiple host machines is probably required to perform large-scale testing with hundreds of virtual clients.

7.3 Traffic Authenticity



(a) q3dm17 Map Overview (b) Gameplay

Figure 7.3: Screenshots of the *q3dm17* Map and Virtual Client Gameplay

Section 3.2.8 on page 30 discussed network generation tools and stated a need for authentic network traffic generation. It means that a tool's generated traffic must have the same transmission patterns as a real human to provide credible test results. A load-generating virtual client needs to produce packets of similar transmission rate and content as a real player. It will otherwise affect the server differently than real players, making the stress-test results not representative of a real-life scenario.

Shin et al [35] performed network traffic analysis to compare the patterns of generated data with the patterns of a real player. This should ideally also have been done for the evaluation of the virtual client implementation, but was left out to limit the scope of this thesis. The reason is that network analysis requires significant amounts of time, and because it is the bot library that primarily defines the traffic patterns through its AI-generated actions. The bot library was however developed to be as close to a real player as possible, making it likely that the produced pattern is close to authentic.

Testing the implementation and playing against the virtual clients show that they behave as real players typically would do. They navigate properly through the map and makes adequate decisions based on their current state. Most tests have been run on the map named *q3dm17* with bot skill (*virtualClientSkill* cvar) set to 4. The map was chosen because it consists of multiple airborne platforms with many *jump pads* and *portals*, requiring a high degree of coordination to not fall off the edges. This is illustrated in figure 7.3a, which is an overview of the map. The virtual clients both navigate and fight in a natural manner on this demanding map, with low rates of failure. Failure usually occurs when attempting to jump using a jump pad or when entering a portal. The virtual client is never completely synchronized with the real server, which makes it sometimes collide with a portal wall or partly miss a jump pad. The implemented synchronization mechanism corrects this in most cases, but it can make virtual clients fall off the map in rare occasions. Figure 7.3b shows a GUI enabled virtual client in action, fighting against 13 other console based virtual clients on *q3dm17*.

7.4 Discussion

This section reflects upon the work and procedures of this thesis. The technical implementation and the entire process of selecting a game are evaluated with suggestions for improvements. An overview of thesis' contributions is provided with suggested areas of use.

7.4.1 Survey on Video Game Features & Qualities

The results of the survey presented in section 4.1.1 on page 37 established the foundation for identifying open source games. Computer scientists from abroad and several Norwegian research and educational institutions were invited to participate, but only 22 felt eligible to respond. It may reduce the credibility of the survey results, as it might not be enough participants to have a representative selection. The top results presented in table 4.1 are however of high relevance with the usage of games in research, discussed in section 3.2 on page 20. This indicates that the results are realistic with demanding requirements, graphics, and good documentation dominating the list. The findings in table 4.2 are hence considered sufficient requirements.

Some survey participants provided various valuable and interesting feedback on both the contents of the online survey and the research topic itself. The most important was regarding the content of the quantitative survey, which was by some perceived as too restricting, and that a large-scale process of qualitative interviews would have yielded better results. This may be due to the specialized research area of the individual scientists that makes it difficult to use a general list for their research. They may however be considered special cases, but it is likely that a separate list for each of the video game utilizing scientific topics would be more useful. This will require work beyond the scope of this thesis.

Further research should include a similar survey of larger scale, contacting computer scientists at research institutions all over the world. Qualitative work such as interviews with scientists experienced in using games in their research could give more detailed results, and lead to multiple lists for the various specialized topics. One can then compare the various lists to identify common properties that may be migrated into a general list.

7.4.2 Open Source Game Evaluation

The proposed list of features and qualities presented in table 4.1 was used to evaluate and identify games eligible for scientific use. Several games were evaluated in section 4.1.3, but only three met the requirements and were proposed as ideal for science. All of these games are far from commercial quality, implying that there currently are no state-of-the-art open source games available. The proposed games are however considered the best alternatives, and should be sufficient for most researchers as they meet most of the requirements in the feature list.

I would ideally like to see more game studios releasing their games as open source. Researchers and industry could achieve mutual benefit from more cooperation on new technologies. Releasing games as open source will require some time and effort from the companies, and is thus usually not perceived as a good investment. Tighter cooperation could alleviate this situation. Researchers could also cooperate with open source projects to achieve a similar goal, but it will likely require more work on the part of the researchers.

Quake III Arena (ioquake3) was eventually selected for the implementation of virtual clients due to its engine features and low latency threshold. Other scientists can compare the proposed games with the feature list to find a game for their research. Their research is then likely to be reproducible and of interest to other scientists because the selected game is open and meets the requirements of other researchers.

The procedure for evaluating and identifying proposed games creates a risk that some potential games may have been missed. The reason is the vast amount of information and code in every open source game project, which makes the procedure require significant amounts of time for every evaluated game. Games were hence evaluated within the time restriction of the project, and evaluation mistakes may have occurred despite the thorough research.

There are other open source games available, and more should be evaluated to obtain a greater list of suitable games. Future re-evaluations are also required as games of the current generation games will become outdated and new options will arrive.

7.4.3 Virtual Client Implementation

Quake III Arena has been converted into a load generation tool by implementing virtual clients. These have been implemented like a local proxy server, by exploiting various features of the engine. A proxy solution was selected because parts of the engine's native bot library was hard coded directly into the server components of the game. This makes it significantly more complex and time consuming to convert the library into a native client feature, which is the alternative. It would also break with Thorn's (2011)[69] engine design principles, discussed in section 2.2.2. The proxy-based solution utilizes *recyclability* and *simplicity* through a lightweight implementation exploiting existing functionality. It allows it co-exist with the native functionality of the original engine, making it easy to merge with updates from the original *ioquake3* project. The simple implementation allows virtual client functionality to be easily triggered, by simply altering a *cvar* configuration value.

Stability is the primary drawback of the proxy implementation. The local shadow server is never 100% synchronized with the real server, and the current implementation doesn't handle all special cases in the engine. Both are due to time constraints of the thesis, as core functionality was prioritized. Most synchronization issues are properly handled by the implemented synchronization routine, discussed in section 5.1.3, but the non-supported special cases lead to undefined behavior.

The implementation currently lacks the support for server messages such as client disconnects and map switches. The latter happens when a round is completed and a new map is loaded on the server. The challenge here is that the local virtual server needs to load the same map as the real dedicated server on a map switch, something the current implementation doesn't account for. It is thus recommended to configure the dedicated server with a huge time and *frag* limit to prevent map switches at run-time. It is not difficult to fix, but will probably require some days of investigation and implementation to make it work properly.

Another issue is the support for multiple game modes. The development was focused on the *Free For All* (FFA) *deathmatch* game mode, which works well. Other game modes are not supported because the virtual client running on the local virtual server only receives entity updates from the real server snapshots, and won't necessarily receive and interpret the team related commands. Significant time with engine investigation and development is expected to resolve this issue.

The largest drawback of the implementation may however be its support for multiple virtual clients. It currently only supports one virtual client per running instance of the software. This requires that multiple instances of the engine and server specific resources are loaded and running simultaneously on a single machine, which both could, and ideally should, be shared among all the virtual clients. Implementing support for multiple virtual clients in one instance of the game will require a significant amount of modifications to both the engine and the client QVM code. The original codebase is hard-coded to support only one client, which makes is not an issue specifically related to the proxy-based implementation. Alternative solutions would also suffer the same restriction. It is however not a huge problem, as section 7.2 proved that a virtual client running in console mode does not consume critical amounts of resources.

A potential better alternative to the suggested two native and proxy-based approaches discussed in this section, would be to develop a virtual client tool that can easily be used with any game. One approach is the learning network generating tools discussed in section 3.2.8 on page 30, which turned out to be a bad idea due to the diversity of video game software. What I however liked was Cho et al's (2010)[10] concept of a configurable *packet analyzing tool* in *VENUS II*, which allows the user to map specific player actions to specific network packets. A similar mapping approach can potentially be used to develop a highly customizable virtual client framework or library that can easily be implemented into any game for full virtual client support. The framework could for instance provide an application programming interface (API) for developers and scientists to map specific actions and behavior to functionality in the engine.

7.4.4 Areas of Use

The three primary findings of this thesis may be used for multiple purposes, both by resources and commercial developers. First is the proposed list of features and qualities, which can be used to identify and evaluate any video game as general-purpose tool for scientific use. All steps of evaluating a game in comparison with the list are documented in section 4.1.2 on page 38.

Evaluating games is however a quite time demanding task, so the thesis proposed three games that are currently eligible for scientific use: *Doom 3*, *Open Arena/Quake III Arena* (ioquake3), and *PlaneShift*. Table 4.2 can be used in correlation with the individual evaluations of each game in section 4.1.3 to select a game that suits your specific purposes. You are then certain to have a game that meets most researchers' requirements, and that allows you to freely publish your research results. It makes the research reproducible, which enhances the credibility of the results. Be aware that the proposed games will eventually become outdated, and new games needs to be evaluated.

The resulting load generation tool of the *Quake III Arena* virtual client implementation should be used to verify previous research where concepts only have been implemented in smaller prototypes. Raaen et al (2012)[50] could for instance implement their unique *LEARS* architecture into *Quake III Arena* and use the virtual client feature to generate load and monitor server performance. They can use the provided shell script files, discussed in section 6.2.2, to make the load-generation and server benchmarking an easy process.

Another area of use is the proxy-based concept itself. Other developers and researchers may use a similar approach if they encounter an engine with the same bot library challenges as *Quake III Arena*. It can be a simple and time sparing approach that gives a lot of functionality with small modifications to the original engine.

Part V

Conclusion

Chapter 8

Conclusion

This thesis provides multiple contributions to academic research on video games. Video games and their use in computer science are thoroughly discussed in chapter 2, indicating a need for an overview of open source games of close-to commercial quality, and a freely available load-generating tool. Chapter 4 identifies a list of researcher's required features and qualities that is used to identify three games eligible for academic use. It validates the selection of *Quake III Arena* for implementing virtual clients, converting it into a freely available load generation tool. The implementation process and instructions of use is documented in chapter 6, and everything is evaluated in chapter 7. This chapter concludes the thesis by providing an overview of contributions and findings, and proposes future works.

8.1 Contributions

This section briefly lists all contributions of the thesis, and provides a summary of the milestones in the following subsections.

The first contribution is an overview of video games and their use in research. Computer scientists were invited to participate in a survey, which resulted in the list of qualities & features to evaluate games for scientific use. The list was used to identify three open source games that are eligible for research. A paper on *Games for Research* was published and presented at LSDVE 2013 covering the findings.

Quake III Arena has been converted into a user-friendly load-generating tool that is freely available. Several software bugs were discovered and reported in *ioquake3*, *SDL*, and *MinGW* during the implementation, some with proposed solutions that have been fixed. The implementation utilizes a proxy-based solution to implement virtual clients, an approach that may be adapted for similar work. This work has been assembled into a small paper, which is submitted to ICEC 2014 and is pending review.

8.2 Games for Research

This thesis has gathered a list of features and qualities that must be met for a video game to be considered suitable for use in computer scientific research. The list has been used to evaluate the suitability of games released with open source code. Our results shows that there are no open source game of commercial quality available, but some are still demanding enough to be interesting for research. A list has been developed of features and qualities that are worth considering when looking for attractive games. The list has been used to identify three potential computer games for research, where one lead to the selection of *Quake III Arena* for the further thesis works.

8.3 Virtual Client Based Load Generation Tool

I have successfully managed to convert *Quake III Arena* into a tool for research on computer games by implementing virtual clients that produce real network traffic and loads the game servers. Migrating the bot library from the server logic into the client logic turns out to be a complex task, so a workaround has been found by implementing a proxy-like solution. It runs a shadow copy of a real server with a bot on the local client that transmits messages between the real server and the bot. I consider it a simple, yet quite elegant solution to a complex problem. There are currently some special case issues with the implementation and room for improvements, but it may still be usable in research like Petlund et. al [48] and Raaen et. al [50] to verify their findings in a large-scale system.

The entire concept of a proxy-based implementation of virtual clients may also be used in future research as a simple way of implementing complex functionality in a large system.

8.4 Further Work

This section summarizes improvement opportunities for the work presented in this thesis, and suggests priorities for further work.

8.4.1 Game Research

Chapter 7 discussed the credibility and criticism of the research that defines the list of required features, leading to the proposed video games for computer science. It may be considered too weak due to the few survey participants and the lack of qualitative research. Comparing the results with the uses of video games in research reveals however that the results reflect the documented use. The findings can hence be considered worth evaluating when a game is to be used for research.

Better and more useful results may however be found by performing similar research of a larger global scale, that includes several more participants and performs qualitative interviews. A list of features and qualities should be identified for each sub-topic discussed in section 3.2, which eventually can be compared to create a list of features common for all research on games. It is also inevitable that this research will become obsolete in time, and new research will need to be done on the topic.

8.4.2 Improving the Virtual Clients

The virtual client implementation in *Quake III Arena* has several improvement opportunities. The most critical is that it is currently only capable of running one virtual client per instance of the software, leading to wasted resources when launching multiple from one host. It requires multiple engine modifications to resolve, and may break the opportunity to retrieve updates for all files from the original *ioquake3* project.

Other improvements revolve around the compatibility of transmitting missing engine features between the local shadow server and the real server. This includes server messages for critical game state changes, such as map switching and team based actions. Details can be found in section 7.4.3 on page 74.

The solution should work well for most scientific uses, but limits the researchers to work with *Quake III Arena*. A highly configurable virtual client software framework or library could potentially be developed for easy implementation of virtual client support in any video game. It is a massive project with a potentially high value for both scientists and commercial developers.

Bibliography

[1] Dmitry Andreev. “Real-time Frame Rate Up-conversion for Video Games: Or How to Get from 30 to 60 Fps for "Free".” In: *ACM SIGGRAPH 2010 Talks*. SIGGRAPH ’10. Los Angeles, California: ACM, 2010, 16:1–16:1. ISBN: 978-1-4503-0394-1. DOI: 10.1145/1837026.1837047. URL: http://doi.acm.org/10.1145/1837026.1837047.

[2] Chris Anley et al. *The Shellcoder’s Handbook, Second Edition: Discovering and Exploting Security Holes*. Second edition. Wiley Publishing, Inc., 2007. ISBN: 978-0-470-08023-8.

[3] Grenville J. Armitage and Sebastian Zander. “Empirically measuring the QoS sensitivity of interactive online game players.” In: *Australian Telecommunications Networks and Applications Conference*. 2004.

[4] Peter C. Bell and Robert M. O’Keefe. “Visual Interactive Simulation - History, recent developments, and major issues.” In: *SIMULATION* 49.3 (1987), pp. 109–116. DOI: 10.1177/003754978704900304. eprint: http://sim.sagepub.com/content/49/3/109.full.pdf+html. URL: http://sim.sagepub.com/content/49/3/109.abstract.

[5] Timothee Besset. *Doom 3 repository on GitHub*. Nov. 2011. URL: https://github.com/TTimo/doom3.gpl.

[6] Timothee Besset. *TTimo / doom3.gpl / Commit history*. Nov. 2011. URL: https://github.com/TTimo/doom3.gpl/commits/master?page=2.

[7] Adriana Braun, Bardo E. J. Bodmann, and Soraia R. Musse. “Simulating Virtual Crowds in Emergency Situations.” In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST ’05. Monterey, CA, USA: ACM, 2005, pp. 244–252. ISBN: 1-59593-098-1. DOI: 10.1145/1101616.1101666. URL: http://doi.acm.org/10.1145/1101616.1101666.

[8] André R. Brodtkorb, Martin L. Sætra, and Mustafa Altinakar. “Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation.” In: *Computers & Fluids* 55 (2012), pp. 1–12. ISSN: 0045-7930. DOI: http://dx.doi.org/10.1016/j.compfluid.2011.10.012. URL: http://www.sciencedirect.com/science/article/pii/S0045793011003185.

[9] John Carmack. *John Carmack Archive - .plan 1999*. Mar. 2007. URL: http://fd.fabiensanglard.net/doom3/pdfs/johnc-plan_1999.pdf.

[10] Chang-Sik Cho et al. "Online game testing using scenario-based control of massive virtual users." In: *Advanced Communication Technology (ICACT), 2010 The 12th International Conference on*. Vol. 2. Feb. 2010, pp. 1676–1680.

[11] Mark Claypool and Kajal Claypool. "Latency and Player Actions in Online Games." In: *Commun. ACM* 49.11 (Nov. 2006), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/1167838.1167860. URL: http://doi.acm.org/10.1145/1167838.1167860.

[12] Mark Claypool and Kajal Claypool. "Latency Can Kill : Precision and Deadline in Online Games." In: *The First ACM Multimedia Systems Conference* (2010).

[13] Misc. contributors. *GitHub ioquake3 Virtual Client Repository*. 2014. URL: https://github.com/stigmh/ioq3/tree/ProxyClient.

[14] Misc. contributors. *Github Official ioquake3 Repository*. May 2014. URL: https://github.com/ioquake/ioq3/.

[15] Daniel Cordeiro, Alfredo Goldman, and Dilma Silva. "Load Balancing on an Interactive Multiplayer Game Server." In: *Euro-Par 2007 Parallel Processing*. Ed. by Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol. Vol. 4641. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 184–194. ISBN: 978-3-540-74465-8. DOI: 10.1007/978-3-540-74466-5_21. URL: http://dx.doi.org/10.1007/978-3-540-74466-5_21.

[16] Atomic Blue Corporation. *About PlaneShift*. Apr. 2013. URL: http://www.planeshift.it/about.html.

[17] Atomic Blue Corporation. *PlaneShift License*. Apr. 2013. URL: http://www.planeshift.it/license.html.

[18] NVIDIA Corporation. *Cuda Toolkit Documentation v5.5*. July 19, 2013.

[19] Valve Corporation. *Doom 3: BFG edition on Steam*. May 2014. URL: http://store.steampowered.com/app/208200/.

[20] Valve Corporation. *Quake III Arena on Steam*. May 2014. URL: http://store.steampowered.com/app/2200/.

[21] P.A. Branch D. Stefyn A.L. Cricenti. "Quake III Arena game structures." In: Feb. 2011.

[22] GameFront. *Game info: Doom 3 | Multi player | Overview*. May 2013. URL: http://doom3.filefront.com/info/Multiplayer.

[23] GameSpot. *Doom 3 Tech Info*. May 2013. URL: http://www.gamespot.com/doom-3/techinfo/platform/pc/.

[24] J.T.T. Goldsmith and M.E. Ray. *Cathode-ray tube amusement device*. https://www.google.com/patents/US2455992. US Patent 2,455,992. Dec. 1948. URL: https://www.google.com/patents/US2455992.

[25] Jason Gregory. *Game engine architecture*. A K Peters, Ltd, 2009. ISBN: 978-1-56881-413-1.

[26] Forum user Hairball. *OpenArena/engine.git on github*. Feb. 2014. URL: http://openarena.ws/board/index.php?topic=4925.0.

[27] Stig M. Halvorsen. *#2183 GCC: command line globbing may affect macro name case sensitivity*. Feb. 2014. URL: https://sourceforge.net/p/mingw/bugs/2183/.

[28] Stig M. Halvorsen. *Bug 6091 - Windows 7/8 - SDL_VIDEODRIVER=directx freezes the OS on breakpoint*. Feb. 2014. URL: https://bugzilla.icculus.org/show_bug.cgi?id=6091.

[29] Stig Magnus Halvorsen and Kjetil Raaen. "Games for Research: A Comparative Study of Open Source Game Projects." In: *Euro-Par 2013: Parallel Processing Workshops*. Ed. by Dieter Mey et al. Vol. 8374. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 353–362. ISBN: 978-3-642-54419-4. DOI: 10.1007/978-3-642-54420-0_35. URL: http://dx.doi.org/10.1007/978-3-642-54420-0_35.

[30] Shawn Holmes. *Focus on Mod Programming in Quake III Arena*. Ed. by André LaMothe. The Premier Press Game Development Series. Premier Press, Inc., 2002. ISBN: 1-931841-56-X.

[31] A S Hornby. *Oxford Advanced Learner's Dictionary of Current English*. Ed. by A P Cowie. 4th edition. Oxford University Press, 1989. ISBN: 0-19-431110-4.

[32] "IEEE Standard for Distributed Interactive Simulation - Application Protocols." In: *IEEE Std 1278.1a-1998* (1998). DOI: 10.1109/IEEESTD.1998.88572.

[33] Michael Jarschel et al. "An Evaluation of QoE in Cloud Gaming Based on Subjective Tests." In: *Innovative Mobile and Internet Services in Ubiquitous Computing, International Conference on* (2011), pp. 330–335. DOI: http://doi.ieeecomputersociety.org/10.1109/IMIS.2011.92.

[34] Tng C. H. John, Edmond C. Prakash, and Narendra S. Chaudhari. "Strategic Team AI Path Plans: Probabilistic Pathfinding." In: *Int. J. Comput. Games Technol.* 2008 (Jan. 2008), 13:1–13:6. ISSN: 1687-7047. DOI: 10.1155/2008/834616. URL: http://dx.doi.org/10.1155/2008/834616.

[35] Yung Woo Jung et al. "VENUS: The Online Game Simulator Using Massively Virtual Clients." In: *Systems Modeling and Simulation: Theory and Applications*. Ed. by Doo-Kwon Baik. Vol. 3398. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 589–596. ISBN: 978-3-540-24477-6. DOI: 10.1007/978-3-540-30585-9_66. URL: http://dx.doi.org/10.1007/978-3-540-30585-9_66.

[36] Raph Koster. *A Theory of Fun for Game Design*. Ed. by Jeff Duntemann. Paraglyph Press, Inc., 2005. ISBN: 1-932111-97-2.

[37] Sam Lantinga. *SDL version 1.2.15 (Historic) downloads*. May 2014. URL: http://libsdl.org/download-1.2.php.

[38] Xiang Liu and Daoxiong Gong. "A comparative study of A-star algorithms for search and rescue in perfect maze." In: *Electric Information and Control Engineering (ICEICE), 2011 International Conference on*. Apr. 2011, pp. 24–27. DOI: 10.1109/ICEICE.2011.5777723.

[39] Peter K. K. Loh and Edmond C. Prakash. "Performance Simulations of Moving Target Search Algorithms." In: *Int. J. Comput. Games Technol.* 2009 (Jan. 2009), 3:1–3:6. ISSN: 1687-7047. DOI: 10.1155/2009/745219. URL: http://dx.doi.org/10.1155/2009/745219.

[40] Phil Maguire et al. "Is Consciousness Computable? Quantifying Integrated Information Using Algorithmic Information Theory." In: (May 2014). URL: http://arxiv.org/abs/1405.0126.

[41] Eddie Makuch. *Valve introduces a new way for you to play PC games in your home*. May 2014. URL: http://www.gamespot.com/articles/valve-introduces-a-new-way-for-you-to-play-pc-games-in-your-home/1100-6419789/.

[42] Ian Millington and John Funge. *Artificial Intelligence for Games, 2nd edition*. CRC Press, 2009. ISBN: 978-0-12-374731-0.

[43] Brice Nzeukou et al. "Personal Gaming." In: URL: http://www.pickar.caltech.edu/news/Personal%20Gaming.pdf.

[44] Lucas Parry. "L3DGEWorld 2.1 Input & Output Specifications." In: *CAIA Technical Report 070808A* (Aug. 2007).

[45] Ioana Patringenaru. *UC San Diego Computer Scientists Develop First-person Player Video Game that Teaches How to Program*. Apr. 2013. URL: http://ucsdnews.ucsd.edu/pressrelease/uc_san_diego_computer_scientists_develop_first_person_player_video_game_tha.

[46] Philipp Fechteler Peter Eisert. "Remote rendering of computer games." In: *International Conference on SIGMAP)* (2007).

[47] Andreas Petlund. "Improving latency for interactive, thin-stream applications over reliable transport." PhD thesis. Unipub, Kristian Ottosens hus, Pb. 33 Blindern, 0313 Oslo: Simula Research Laboratory / University of Oslo, Dec. 2009. ISBN: ISSN 1501-7710.

[48] Andreas Petlund et al. "TCP enhancements for interactive thin-stream applications." In: *18th International NOSSDAV Workshop*. NOSSDAV '08. Braunschweig, Germany: ACM, 2008, pp. 127–128. ISBN: 978-1-60558-157-6. DOI: 10.1145/1496046.1496081. URL: http://doi.acm.org/10.1145/1496046.1496081.

[49] ioquake3 project. *Download ioquake3*. May 2014. URL: http://ioquake3.org/get-it/.

[50] Kjetil Raaen et al. "LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition." In: *41st International Conference on Parallel Processing Workshops*. ICPPW '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 382–389. ISBN: 978-0-7695-4795-4. DOI: 10.1109/ICPPW.2012.55. URL: http://dx.doi.org/10.1109/ICPPW.2012.55.

[51] Eoin Redahan. *Flooding the system - improved flood simulation technology*. Mar. 2012. URL: http://www.iom3.org/news/flooding-system-improved-flood-simulation-technology.

[52] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach, 3rd edition*. Pearson Education Limited, 2010.

[53] Forum user sago007. *Gitorious*. Feb. 2014. URL: http://openarena.ws/board/index.php?topic=4309.50.

[54] Fabien Sanglard. *Doom3 BFG Source Code Review*. May 2013. URL: http://fabiensanglard.net/doom3_bfg/.

[55] Fabien Sanglard. *Doom3 source code review*. June 2012. URL: http://fabiensanglard.net/doom3/.

[56] Fabien Sanglard. *Fabien Sanglard's Website*. May 2014. URL: http://fabiensanglard.net/.

[57] Fabien Sanglard. *Quake 3 Source Code Review: Architecture*. June 2012. URL: http://fabiensanglard.net/quake3/.

[58] Shacknews. *John Carmack and id Software's pioneering development work in 3D game engines recognized with two technology Emmy awards*. July 2008. URL: http://web.archive.org/web/20080705061409/http://www.shacknews.com/docs/press/010710_id_carmack_emmys.x.

[59] Shacknews. *Quake 3 Source Code Released*. Aug. 2005. URL: http://www.shacknews.com/article/38305/quake-3-source-code-released.

[60] Kwangsik Shin et al. "Online Gaming Traffic Generator for Reproducing Gamer Behavior." In: *Entertainment Computing - ICEC 2010*. Ed. by Hyun Seung Yang et al. Vol. 6243. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 160–170. ISBN: 978-3-642-15398-3. DOI: 10.1007/978-3-642-15399-0_15. URL: http://dx.doi.org/10.1007/978-3-642-15399-0_15.

[61] Roger D. Smith. "Essential Techniques for Military Modeling and Simulation." In: *Proceedings of the 30th Conference on Winter Simulation*. WSC '98. Washington, D.C., USA: IEEE Computer Society Press, 1998, pp. 805–812. ISBN: 0-7803-5134-7. URL: http://dl.acm.org/citation.cfm?id=293172.293309.

[62] id Software. *id History*. Mar. 2013. URL: http://www.idsoftware.com/business/history/.

[63] Erlend Lånke Solbu. *Gameshow i forelesningssalen*. Feb. 2014. URL: http://www.nrk.no/viten/gameshow-i-forelesningssalen-1.11516268.

[64] Håkon Kvale Stensland et al. "Tips, Tricks and Troubles: Optimizing for Cell and GPU." In: *The 20th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2010)*. Ed. by Dick C. A. Bulterman. ACM, June 2010, pp. 75–80. ISBN: 978-1-4503-0043-8.

[65] Forum user Stigmha. *OpenArena Forum: Re: OpenArena/engine.git on github update #2*. Feb. 2014. URL: http://openarena.ws/board/index.php?topic=4925.0%5C#msg50278.

[66] Avneesh Sud et al. "Real-time Path Planning for Virtual Agents in Dynamic Environments." In: *ACM SIGGRAPH 2008 Classes*. SIGGRAPH '08. Los Angeles, California: ACM, 2008, 55:1–55:9. DOI: 10.1145/1401132.1401206. URL: http://doi.acm.org/10.1145/1401132.1401206.

[67] Kongsberg Defence Systems. *KONGSBERG has signed a contract for upgrade of the Royal Norwegian Navy's (RNoN) PROTEUS simulator infrastructure*. Mar. 2014. URL: http://www.kongsberg.com/en/kds/news/2014/march/proteus-contract-with-royal-norwegian-army/.

[68] Crystal Space Team. *Crystal Space official site*. Apr. 2013. URL: http://www.crystalspace3d.org/main/Main_Page.

[69] Alan Thorn. *Game Engine Design and Implementation*. Foundations of Game Development. Jones & Bartlett Learning, LLC, 2011. ISBN: 978-0-7637-8451-5.

[70] James Tulip, James Bekkema, and Keith Nesbitt. "Multi-threaded Game Engine Design." In: *Proceedings of the 3rd Australasian Conference on Interactive Entertainment*. IE '06. Perth, Australia: Murdoch University, 2006, pp. 9–14. ISBN: 86905-902-5. URL: http://dl.acm.org/citation.cfm?id=1231894.1231896.

[71] Various forum users. *The Official OpenArena Forums: Does the engine even build any more?* May 2014. URL: http://openarena.ws/board/index.php?topic=4900.0.

[72] Luis Valente, Aura Conci, and Bruno Feijó. "Real Time Game Loop Models for Single-Player Computer Games." In: (2005). URL: https://www.ssugames.org/pluginfile.php/1026/mod_resource/content/1/2005_sbgames.pdf.

[73] Jean Paul van Waveren. "The Quake III Arena Bot." MA thesis. the Netherlands: Delft University of Technology, June 2001. URL: http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren_Jean-Paul_van/thesis.pdf.

[74] Wikipedia. *Id Tech 4 — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-April-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=Id_Tech_4&oldid=548680739.

[75] Wikipedia. *List of open-source video games — Wikipedia, The Free Encyclopedia*. [Online; accessed 2-April-2013]. 2013. URL: http://en.wikipedia.org/w/index.php?title=List_of_open-source_video_games&oldid=548300619.

[76] David Winter. *PONG-Story: The site of the first video game*. Feb. 2014. URL: http://www.pong-story.com/intro.htm.