Response time in games: Requirements and improvements

Kjetil Raaen

Abstract

Many computer games, both modern and historical, focus on action. Success in these games relies on hand-eye coordination and fast reaction time. However, the player's reactions are not the only reactions that need to be fast. The computer and if in use, the network, should also be fast enough not to slow down the game. If the system becomes the bottleneck for reactions in a game, the experience for the player will degrade. Several current trends motivate this thesis. Users want their computers to react fast. This is especially important in fast paced games. At the same time, players want to interact with increasing numbers of others in the games. To handle these requirements, we have increasingly parallel computers available, and cloud services proliferate.

All current games have the ability to report network latency and frame-rate, used as a proxy for local delay. These metrics do, however, not capture all delay experienced by the gamers. In this thesis we measure the full delay from user input to response on screen. We develop an external measurement setup that captures both input and output and measures the time between them. We find that delays are highly variable, from 33 to 170 ms. While current research on the effects of delay ignores local delay, we find that the magnitude of the local delay is significant, and should be considered.

To assess the impact delays may have, we look at human sensitivity to delays. This has been investigated in many ways before, but conclusions diverge. To improve the understanding of the topic, we present a new set of basic experiments looking for the point where delay becomes detectable to the participants. Specifically, we look at delay between a motor input by the user and visual output on a screen. By varying the delay, we seek to determine a threshold of consciously detectable delays. Our results show first of all a large individual difference in this metric. Some participants can detect delays close to our lower experimental threshold of 50 ms, while others seem unable detect delays as high as our upper experimental threshold of 500 ms. Our questionnaire on the background of the participants does not reveal any explanation for the individual differences. The median detectable delay is around 200 ms.

Further, we focus on important technical challenges of improving delay in games. We investigate how much delay cloud services add to a game server, because cloud services are becoming a popular way of deploying online services. We find that the current services are not ideal for action games, but can work well for games that are more tolerant of delays.

Some genres of games are popular because they allow large numbers of players to gather in a shared area of the virtual world. Supporting such gatherings of players without compromising response time requires novel server architectures. We propose and implement an architecture that parallelises server load efficiently by relaxing some consistency requirements, in order to increase scalability. Our proposed architecture allows large numbers of players to gather while distributing work evenly among many CPU cores.

We hope researchers and developers working on games will benefit from the results of this thesis. Understanding both the limitations of human perception, as well as details of software and hardware architecture, is critical to properly address the bottlenecks of game scalability.

Acknowledgements

Completing a large work like this is not possible alone, and I have many to thank. My advisors, Andreas Petlund, Pål Halvorsen and Carsten Griwodz have helped me all the way with experience, knowledge and inspiration. These have also been co-authors of some of the papers presented here. I would also like to thank the others I have worked with as co-authors on the papers. They have all contributed far beyond the papers their names are on. Håvard Espeland Håkon K. Stensland, Ragnhild Eg, Tor-Morten Grønli and Ivar Kjellmo have been excellent teammates and have made this work feel far less lonely.

Without *Westerdals – Oslo School of Arts, Communication and Technology*, which was *NITH* when I started I would not have gotten the opportunity at all. I would especially like to thank Bjørn Hansen and Eivind Breivik for taking their chances with a former game programmer who wanted to go into academia. This institution have also provided me with other colleagues who have given support and motivation when needed. Among these I would like to mention Trude Westby for bringing me books and Wanda Presthus for discussions and support in writing as well as Stig Magnus Halvorsen and Marius Brendmoe for close cooperation in different phases of the process. *Simula Research Laboratories* and all the colleagues there have also been important support in the process.

Lastly, but by no means least, I would like to thank my friends and family who have supported me and kept me happy throughout all this. My parents Thoralf and Elin and their love, support, encouragement and constant willingness to answer the question ”Hvorfor?” (”Why?”) built the foundations for me as a person as well as for my education. My brothers, Håkon and Gunnar give me both support as well as much needed rivalry. During the work on this thesis I have also gained a wife and a dog, and I wold very much like to thank Sunniva and Aurora for always being there for me with warmth and love.

Contents

List of Figures

List of Tables

Part I

Overview

Chapter 1

Introduction

Running through the damp, dark tunnel you know the enemy is not far behind. If you can just get through that old door over there, you will be safe for now. You pull the door open carefully, making sure your gun goes first. Suddenly, a massive spider drops down from the mouldy ceiling. You pull the trigger, but nothing happens, not even a click. Then, the spider is upon you. By the time the gun goes off, you are already eaten.

Fortunately, this was only a game. But in games, a delay of even a fraction of a second can make the difference between winning and losing. This work investigates how small the delay needs to be to avoid such situations, as well as how to make it shorter. We have worked on how to parallelise game servers more efficiently and investigated whether the cloud services and large non-uniform memory access systems can be used for running servers.

This chapter defines the work in more detail by presenting the motivation and detailing the questions we have asked, as well as the scientific methods used to answer them.

1.1 Motivating trends

Computer games are maturing as a cultural expression. Real-time games can be traced all the way back to the forties when radar developers first started playing with their screens (Goldsmith and Ray, 1948). In the early seventies, games became a commercial product starting with the famous game *Pong* from Atari (Mennie, 1976). In the more than forty years since then, games have become an important form of culture, for fun and even serious artistic expression. The technology involved has developed from circuits containing a few transistors to relying on multiple components with billions of transistors in each. The graphics quality of games is approaching photorealistic, while scenes and stories have become huge and increasingly complex.

Such increasing complexity naturally introduce problems and challenges. We focus on the challenge of how to keep interactions responsive despite the increasing complexity. This is especially relevant in fast paced action games. This difficulty is exacerbated by game designers' desire to scale these games to what they call *massively multiplayer*, providing simultaneous support for thousands of players. The main opportunity we study to meet these challenges is the steadily increasing levels of parallelism available in computer systems.

While games are the original motivation and central theme of this thesis, the results and conclusions can be applied in a much wider context of realtime human-computer interactions.

Among others, cooperative work environments on the Internet and teleconferences both deal with similar challenges.

1.1.1 Responsiveness in interactions

Engineering practice since the early days of motion pictures assume that less than 50 frames per second is enough for motion to appear fluid. Cinema projections still use 24 frames per second. At the same time, many who play action games are convinced that they need much higher frame rates to be competitive in games. Some players tune their games to barely acceptable visual quality working towards hundreds of frames per second. To understand the differences between frame-rate in motion pictures and in games, we need to look at the concept of *responsiveness*.

When interacting with computers, users expect the system to respond within some limited time. Exactly zero delay is computationally impossible, so users have to accept some delay. How much delay is acceptable depends on the nature of the task. If the task is downloading a software suite from the Internet, most accept minutes, even hours of delay provided some feedback on the progress of the operation. At the other extreme, a player firing a gun in a fast-paced action game is much less patient, expecting a response within a fraction of a second.

As computers become more powerful, the added power is utilised to improve quality and increase the complexity of all software, and games are no exception. However, this added complexity does have some downsides. Adding network communication as well as improving graphical quality could introduce more delay than the improved performance of the hardware subtracts, resulting in a slower overall experience. Figure 1.1[1] shows the most relevant sources of delay in user interaction. These include input-output units such as screens, mice and keyboards, as well as software components, such as operating systems, drivers and the application itself. If the game is networked, network infrastructure adds another series of delays.

1.1.2 Action games

Some aspects of the earliest games are still present in many modern games. Among these are the reliance on the player's reaction time and motor skills. Games where this is an important aspect are often simply referred to as *action games* (Egenfeldt-Nielsen et al., 2009).

Action games rely heavily on reaction time, creating a unique challenge in engineering: While most software only need to respond before the user gets *impatient*, action games need to be able to respond before users even *notice any delay* at all. If a system require fast responses from the players, but players notice that the system does not respond immediately to their commands, they could feel that the game does not work well or is unfair. Some have even found indications that delays in games affect players even when it is *lower* than what they are consciously aware of (Chen and Lei, 2012). These results should all be considered when building and deploying games.

Thus, action games create unique challenges. Clients, networks and servers all need to respond very quickly to avoid long delays. Games are also complex systems which without careful optimisation can easily become slow.

[1] Illustration by Ivar Kjellmo.

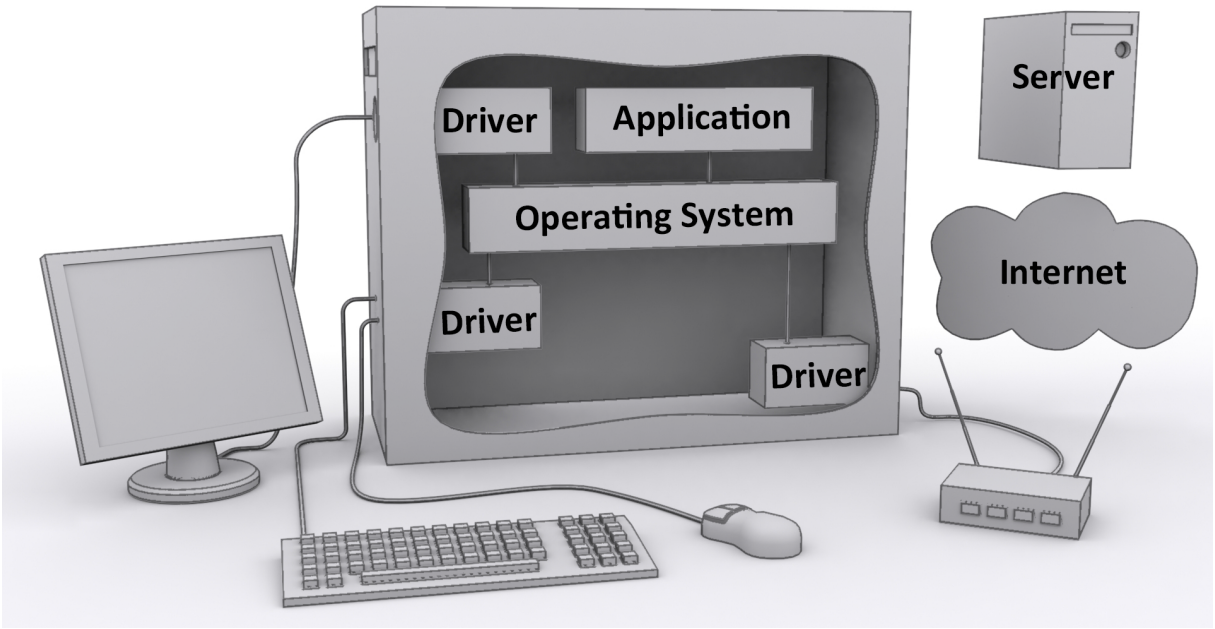


Figure 1.1: Potential sources of delay. Not all of these are relevant in all interactions.

1.1.3 Massive games

For a while now, a class of games called *Massively Multiplayer Online* (MMO) games have been popular. This expression seems to have no generally accepted definition, despite its frequent use. For the purpose of this work, we define an MMO as follows any game that fulfil the following criteria:

- The game allows at least a thousand players to be online at the same time, sharing the same world.
- The game supports at least a million players over time.
- The game has a centralised server architecture.
- Each player has one or more avatars saved on the centralised server.

Games with these properties present a unique set of technical challenges. The large total number of players requires large and sophisticated architectures to handle. However, this is not unique, many other server applications face similar problems. However, most other applications do not allow clients to communicate in realtime on a many-to-many basis. The requirement that all users, or at least a large proportion, should be able to communicate with each other in realtime while running heavy individual calculations creates some unique challenges.

1.1.4 The growth of parallelism

Software developers have adapted to and come to expect exponential growth in computing power over the last few decades. This growth has led to a long series of innovations that, in many ways, have changed the world. The exponential growth of circuit complexity was first

described by Moore (1965) while working on early integrated circuits and has continued since. A common misunderstanding of Moore's law is that it describes growth in *performance*. Moore only refers to growth in *complexity*. This distinction is important; translating increased transistor count to increased actual performance is difficult. Recently, the growth in single-threaded performance has more or less levelled off. Now, most performance gains are based on increasing the number of processor cores in a chip (Dally, 2009).

Utilising these cores requires deep changes in how we make the software. Amdahl's law (Amdahl, 1967) states that the speedup achieved by parallel processing is limited by the time needed for the sequential fraction of the program. Reducing this sequential fraction is one of the main challenges to continued performance improvements. In games, this often requires rethinking architectures at a fundamental level. However, even a parallel algorithm requires synchronisation to access memory, caches and input/output devices, again reducing performance.

1.1.5 Cloud computing

Heavily influenced by earlier concepts such as *grid computing* and *utility computing* (Foster et al., 2008), *cloud computing* (Vaquero et al., 2009) is a relatively recent paradigm for computing architecture. Instead of owning servers and other infrastructure, or renting them on long term contracts, cloud computing allows clients to rent infrastructure on a hour-by-hour basis. Cloud services provide APIs that allow the software itself to add or remove capacity as needed. This gives much better flexibility to clients than ownership or long-term rents. Providers of online games could benefit from this, but it is not clear if the infrastructure is suitable for applications sensitive to delay and jitter (Barker and Shenoy, 2010).

1.2 Problem statement

From section 1.1, we identify a need for more knowledge about latency in games. This question has many aspects, including technical, perceptual and even artistic. The technical questions formed the original motivation for this work; however, some steps are needed to work effectively on improving technology. Hence, we address the following questions:

1. "How much delay is there currently in games?"

   Maybe somewhat surprisingly, the actual amount of delay present in current games and systems is not well known. Developers focus on the application layer while many other sources of delay are possible in a computing system. Without such knowledge results from user tests can not be compared across studies. Hence, measuring and understanding the full delay is important to evaluate both previous research and our own results.

2. "How fast should game systems respond to avoid deterioration of player experience?"

   This question is not a question of technology, but rather of perception. However, it is closely tied to technology. Different games might lead to different perception of delay, and even within one game there might be different types of delay. Motivated by action games, we are particularly interested in the lower limits of perception of delay. Answering this question is important as a benchmark for technical improvements. A system that

can consistently be faster than the users require can be considered good under the tested conditions. Previous work give conflicting results, and rarely investigates the participants' background.

3. "Are cloud services responsive enough to run game servers?"

   Currently, cloud services are considered the ideal for deploying most new services. However, games have different characteristics than other server loads, especially regarding response time. Because cloud services in general run on virtual machines, performance characteristics might not be the same as native servers. Using these for game services is only viable if response time can be kept low.

4. "How can parallelism be better utilised in game servers for massive multiplayer?"

   Games can scale in many dimensions. For massive multiplayer games, one of the most important dimension is player density: The number of players that can fit into a given area of the game world without compromising performance. Such scaling is also technologically challenging. Utilising parallelism is necessary to go beyond current densities, but it creates many challenges. In dense gatherings, all participants need to be continuously updated with information about what all others do.

1.3 Research method

In this section, we introduce the methodologies we chose to address the questions presented above.

1.3.1 Literature survey

When gathering information about a new field, reviewing existing research is crucial. A systematic literature survey might also be of interest to others. Three basic steps are part of a literature review. First, we identify relevant literature. Second, we summarise the work and identify common trends. Last, we discuss the content and attempt to draw conclusions. The paper presented in appendix I uses this method to investigate how fast game systems should respond.

1.3.2 Natural science in information technology

Not all knowledge about computers and the software running on them can be gained by studying their design. Multiple factors contribute to this. The massive complexity of current systems is one, and trade secrets is another. More fundamentally, a computer that *interacts* with external processes during processing cannot be modelled using traditional algorithmic concepts (Wegner, 1995).

Natural science seeks to explain how things are. The term *natural* science does not preclude studying artificial phenomena. The methods of natural science are used to study how organisations work or the properties of synthetic compounds (March and Smith, 1995). We use the same methods for studying computer systems. Accordingly, when we investigate how much delay there is in current systems, we employ experimental natural science on computer systems.

By running a system under controlled conditions and systematically recording results we get information about the behaviour of the system that would otherwise be unavailable.

1.3.3 Design science in information technology

Where natural science aims at producing theoretical knowledge, design science aims at producing useful artefacts. Such artefacts are then assessed to see if they work and are improvements on current technology. The ACM refers to this type of science as the *design paradigm* of computer science (Denning et al., 1989).

Much work in design science can be evaluated on relatively simple, objective metrics, and we can create these using simulations or emulation. We applied this concept when we worked on new concepts for improving games servers.

1.3.4 Perceptual psychology

To investigate the impact and effects of delay on users and players we need empirical data from human participants. The discipline of *psychophysics*, as described in "Psychophysics: The fundamentals" (Gescheider (1976)), focuses on measuring the relationships between psychological *sensations* and physical *stimuli*. A central concept in this discipline is the *sensory threshold*, defined as the minimum magnitude of a stimulus needed to produce a sensation. Usually, this threshold refers to a measurable physical property that varies in intensity. While we want to measure time intervals, the analogy to a sensory threshold is good; we measure the sensory threshold for the sensation of time. Of the standard experiment types from psychophysics, we have used the *method of constant stimuli* and an adapted version of the *method of adjustment*.

Applied to games, this sensory threshold determines if the players *notice* the delay. The wider question about what delays actually *influence* players need a different methodology, and is outside the scope of this thesis.

1.4 Main Contributions

This work presents multiple contributions all focused on the same goal of improving delay conditions in games. This section refers to the research questions described in section 1.2.

To answer question 1, we present work on measuring how much delay is present in current games. It turns out that traditional metrics such as frames per second or internal timings are not enough to get a complete picture of the delays present. From the time a player clicks a button to fire until the weapon goes off on the screen, multiple things have to happen; only some of those are under the control of the game software itself. Reported performance metrics in this situation are frame-rate and in the case of networked games, network latency. These do not capture all delays. Components of hardware and software not under the control of the programmer add delays which are impossible to capture in software. These delays are also heavily dependent on the exact system and configuration the game is running on.

To get around this limitation, we built an external setup to measure these delays. This setup is itself a contribution because previous systems to measure delay have been more complicated or less accurate. Using this setup we gathered data from different systems. In general, our

results show that local delays are longer than traditionally assumed. For realistic situations, we see local delays ranging from 30 ms best case all the way to 200 ms. If the game is networked, network communication adds additional delays.

Regarding question 2, we investigated how short delays had to be to not affect the experience of players. Previous research on effects of delay in games focused mainly on network latency and other sources of delay are mostly ignored. From our work on actual delay in games, we know that other sources of delay can often be as long if not longer than network latencies. Therefore, we investigated perception of delays in very simple interactions, hoping to find some basic perceptual limits and eliminating many error sources from the system compared to previous research.

We found that results are highly individual. It seems that some people can detect delay well below 100 ms while others barely notice three times this value. Somewhat surprisingly, we found no correlation between experience in playing computer games and the threshold for detecting delays. There is a significant overlap between delays that are detectable by players and actual delay present in games: Some players can easily notice delays of many game systems. This situation is not ideal because it impacts the quality of experience of the players as well as their performance in the game.

Cloud services are becoming the norm for deploying servers for any project. To answer question 3, we investigated if these have the characteristics needed to deploy game servers. Particularly, we looked at how stable performance the cloud services delivered at millisecond resolutions. Our results indicate that cloud servers are not yet ready for hosting games with realtime requirements, because the performance is not predictable enough.

Massive multiplayer games, as described in question 4, allow potentially thousands of players to play in a shared world. In reality, only relatively few players can actually interact with each other at any one time. In real game implementations, players are split by artificial barriers to allow load distribution among servers. We propose a game server architecture that alleviates these limitations. By relaxing some consistency constraints and splitting the work into small tasks, we are able to run some hundred players without artificial partitioning.

1.5 Outline

The rest of this text is organised as follows:

Chapter 2, Games and response time: Defines relevant terms and describes the context this work builds on.

Chapter 3, Measuring delay in games: Introduces a system for measuring actual delay in computer systems used to play games, as well as some data gathered using the setup. Further, it discusses some consequences of these delays.

Chapter 4, Limits for observable delay: Presents a series of experiments to establish how long delays from input to result can be before users notice, and compares them to information gained in the previous chapter.

Chapter 5, Game servers in the cloud: Investigates if cloud services are suited for game servers.

Chapter 6, Parallelising game servers: Describes a novel architecture for a parallel game server to minimise delays under high load.

Chapter 7, Paper and author's contributions: Describes each paper included in this thesis along with lessons learned and the author's contributions to this paper.

Chapter 8, Conclusion: Summarises and concludes this work and presents ideas and concepts for further study.

After these chapters, all papers are included as section 2.

Chapter 2

Games and response time

When working on response time in games, whether with the aim to measure it or improve it, we need some terminology and clear definitions of terms. This chapter aims to introduce important background regarding the concept of delay in games, and how it might affect players.

2.1 The importance of delay

Modern computing systems are used interactively. However, there is always a delay between the time the user sends input and the result appearing on the screen. This delay is commonly termed *lag*. In the field of human-computer interaction, Seow (2008) defines *instantaneous* response as the system responding fast enough that the user is not aware of any delay. For graphical controls and other interactions that mimic the physical world, Seow recommends instantaneous response time. In games, most interactions either mimic the physical world or are graphical in nature, indicating that responses should be instantaneous. Also, delays annoy players and thereby give a negative impression of a game. If the players are not aware of delays, they will likely not be annoyed. While delays shorter than this may be imperceptible, they can still affect user performance, for instance through increased stress levels (Chen and Lei, 2012).

Response time is considered a critical factor for the *Quality of Experience* (QoE) in networked games. For local games, on the other hand, this metric is often ignored. Achieving sufficiently low local response time has rarely been considered a problem. The dominating factor of response time in networked games and interactive worlds is traditionally assumed to be network latency; hence, earlier work is mostly focused on network metrics. Conversely in local games, lacking any communication, response time is only bound by computation, which is considered a known factor when creating the game and can thus be compensated for. However, there is very little evidence for this assumption, and if it does not hold, much research will have to be reevaluated.

During optimisation and quality control of a game, developers currently rely on subjective tests of the current game, as well as some rules-of-thumb. If developers had a table of delay based on empirical data from different games, they could look up the game that most closely matches their game and use that result as their design goal. This would save time and effort because long delays could be diagnosed by simple technical measurements rather than being discovered in much later user testing.

2.2 Genres of games

When working on and discussing responsiveness in games, it is important to be aware of classifications of games. A clear classification allows comparing similar games. Most individual contributions on response time in games focus on a specific class, or genere, of game, which is likely to give more consistent results. Conversely, numbers from one genre might not be applicable to a game from another genre. Therefore, we introduce one of the most widespread classifications of games, which we use in this work.

2.2.1 Gameplay style

Claypool and Claypool (2006) describe a whole set of game types and situations and recommend latency limits, dividing games into three broad categories. It is assumed that each of these classes of games has different latency requirements. However, there are large variations within each genre. Even individual games may have very varied content and interactions.

First Person Avatar

This term describes games where the player controls an avatar, and the game is displayed from the point of view of that avatar as if the player sees *through the eyes* of the avatar. Input directly controls what the character does, such as moving or steering. The most common variants of this genre of games are the *racing games* and *first person shooter* (FPS).

In racing games, the player controls cars or other vehicles and tries to win a race. The screen usually shows the controls of the vehicle and the track as seen through the window of the vehicle. Well known examples are the "Need for Speed" (Electronic Arts, 1994) series and the "Forza Motorsport" (Microsoft Studios, 2005) series.

First person shooters focus, as the name implies, on shooting. The screen shows the world as seen from the point of view of the player characters combined with the weapons in their hands. Some attempt to be realistic combat simulations, such as the "Medal of Honor"(Electronic Arts, 1999) series, while others have a looser base in reality such as the "Unreal Tournament" (Epic Games, 1999) series. Some first person shooters support enough simultaneous players to be termed *Massive Multiplayer First Person Shooter* (MMOFPS).

Claypool and Claypool consider first person avatar games to be the most demanding in terms of latency requirements, recommending total latencies less than 100 ms.

Third Person Avatar

These games are also based on a single avatar controlled by the player, but the avatar is seen from the outside. Input often gives characters *instructions*, such as *move there* or *attack this* rather than direct control. The camera usually follows the avatar. Some games can switch between first and third person view to accommodate player preferences. Popular examples include "The Elder Scrolls" (Bethesda Softworks, 1994) series and the "Diablo" (Blizzard Entertainment, 1996) series.

The most studied variant in this group is the *Massively Multiplayer Online Role Playing Game* (MMORPG) game. Among these, the best known is "World of Warcraft" (Blizzard

Entertainment, 2004).

Claypool and Claypool consider third person avatar to have intermediate latency requirements, and recommends latencies lower than 500 ms.

Omnipresent

In these games, the player does not control a specific avatar, rather multiple elements in a large game world. The player can look at any part of the game world at any time. The most popular variant of this is the *Real Time Strategy* (RTS) game, with well known examples such as the "Starcraft" (Blizzard Entertainment, 1998) series and the "Command & Conquer" (Electronic Arts, 1995) series. Other variants include *turn based strategy* games, including the famous "Civilization" (MicroProse et al., 1991) series of games.

Claypool and Claypool claim these games have very lax latency requirements, with up to 1000 ms acceptable latency.

2.2.2 Massive Multiplayer Online Games

Massive Multiplayer Online Games is a catch-all term for games supporting hundreds of concurrent players. We use the definition described in section 1.1.3. MMOGs are particularly interesting cases, and the games themselves, as well as the technology behind them, have received significant attention from computer scientists. In addition to the requirements regarding latency, these also need to support large numbers of players. Yahyavi and Kemme (2013) put it this way: "One of the main attractions of these games lies in the number of players that participate in the game. The more players that are in the game world, the more interactive, complex and attractive the game environment will become."

2.3 Scaling games

We saw in section 2.2.2 that scaling games to allow larger numbers of players is interesting from the game developer perspective. Müller and Gorlatch (2007) identify what they term *scalability dimensions*. These are different dimensions games can scale along. They identify three important ways in which large-scale online games can scale.

> - *Overall number of simultaneously active users* needs to scale while all these users should be able to interact.
> - *Game world size* should scale. This increases memory requirement to store the larger areas, as well as processing power required to run all automatic activity in the areas.
> - *Player density* should scale. Players are often drawn to already existing action, creating an escalating spiral of increased player density, which servers ideally should support.
>
> (Müller and Gorlatch, 2007, p. 82)

The importance of the third dimension is underlined by Chen and Lei (2006). While investigating player movements, they find that players tend to clump together rather than spread

out evenly across the game map. They conclude: “The analysis reveals that the dispersion of players in a virtual world is heavy-tailed, which implies that static and fixed-size partitioning of game worlds is inadequate.” (Chen and Lei (2006))

This tendency of players to gather in small spaces limits the effectiveness of any implementation relying purely on spatial partitioning. Thus, we need to find a solution that allows players in the same area, even at the same exact point, to use different processing resources.

2.4 Game communication modes

In addition to the genre of the game itself, it is important to distinguish how games utilise networks. Currently, three models are actively in use: *local games*, *client-server games* and *cloud gaming*. A fourth, *peer-to-peer games* has received some academic attention, but it is not widely used because the challenges, especially in security, have never been solved.

2.4.1 Local games

The simplest way to make games is for each player to play alone on their own computer. While variations of cooperation or competition currently are common in games, players sometimes still play alone on their own computers or game consoles. In terms of delay, single player games are predictable and easy to analyse. However, even in these games, information has to travel through multiple steps between input and output, as shown in figure 2.1. Signals from input devices go to through drivers, then to the operating system before being sent to the application. Output signals follow a similar path.

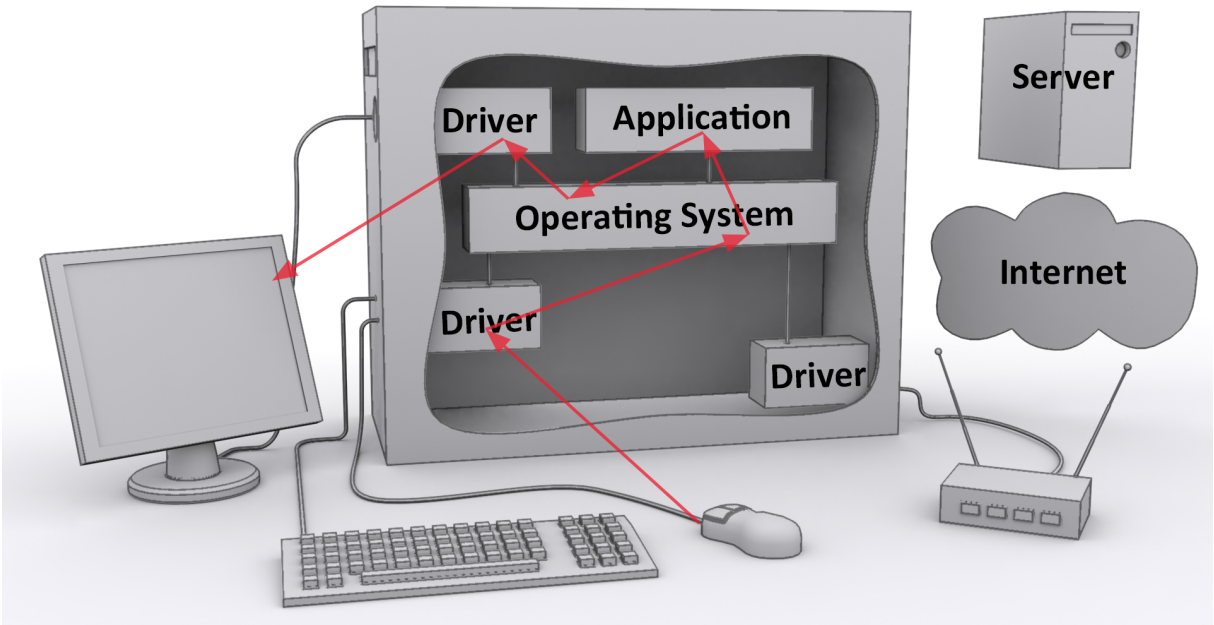


Figure 2.1: Path of information in local games.

2.4.2 Client-Server games

Conceptually, a *client-server architecture* means that the client transfers data to the server; the server does necessary processing and sends back the result. The client then renders and displays the scene using parameters received from the server. However, in order to alleviate latency issues and network load, strict adherence to the client-server pattern is rare except in purely experimental games.

Client-server games are all designed from the ground up to handle network latency, deviation or displacement in timing and *jitter* (defined as variations in delay). By employing various prediction techniques and allowing a looser state consistency between different players, games are able to alleviate, or in some cases completely isolate, the players from the effects of network and server latency. These techniques include allowing the client to do some calculations locally, thus making feedback much quicker. Usually referred to as *client-side prediction* (Bernier, 2001), this solution will, at its simplest, run exactly the same code on the client as would run on the server. Whenever an update is received from the server, the client state is reverted to conform to this data. Furthermore, instant effects, such as weapons firing, are executed immediately on the local client to give the player a feeling of responsiveness. To allow players to hit where they aim, and other quasi-immediate effects, the system needs to predict where other players are at a given moment, increasing the complexity of the system.

2.4.3 Cloud gaming

The concept of *cloud gaming* represents *Software as a Service* (Vaquero et al., 2009) where the software is a game. Delivering games in this way presents some unique challenges, possibly including different requirements on the network latency. The cloud, or remote server, performs all game logic and rendering tasks and generates the output to display in a video. The client closely resembles a traditional *thin client*, which simply forwards commands from the user to the server, and displays video from the server. In this scenario, requirements on client hardware is very low. As a tradeoff, the requirements on the network link are significantly higher. There are at least two clear reasons for this. First, transferring high definition video requires significantly higher throughput than simple control signals. Client-server games require 10 - 60 kbps (Petlund, 2009, p.10), while high resolution video streaming requires 2 - 8 Mbps Pourazad et al. (2012). That is, approximately a factor of a hundred higher network throughput. Secondly, none of the techniques to alleviate network delay described above can be applied.

Figure 2.2 shows the path for information from user input to screen in cloud gaming. Here, in addition to the steps needed in a local system, signals have to pass through network hardware, network software as well as the server before results are visible to the player. Services such as Nvidia Grid (Nvidia, 2015) provide a wide array of games in all generes using this type of infrastructure.

Latency compensation is more difficult in this scenario, because such techniques require local computing power, which is exactly the requirement cloud gaming aims to avoid. Recent work has shown that latency compensation in cloud gaming is possible, at the cost of increased bandwidth usage. Lee et al. (2014) suggest that the server sends the results of multiple user actions simultaneously, and the client chooses which to display based on user actions. Thus, for every frame, the server renders one where the player has fired a gun, and one where the

gun is not fired. The client displays the frame with the correct outcome based on wether the player pushed the trigger button. Although the authors suggest various improved compression schemes for this system, bandwidth requirements are still significant.

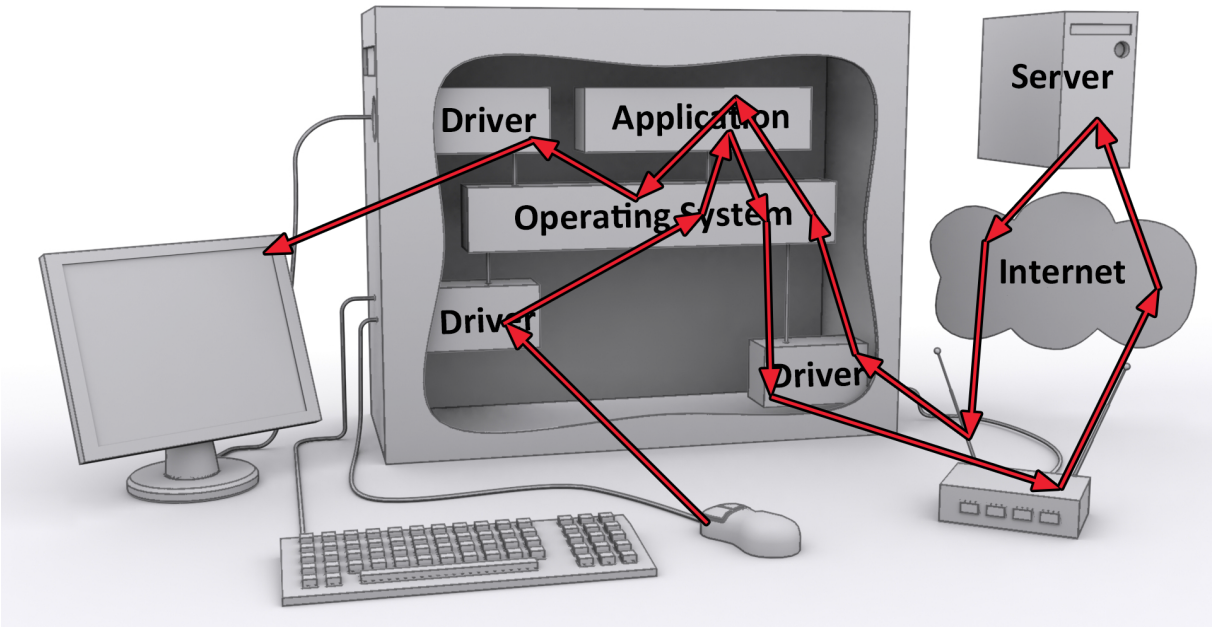


Figure 2.2: Path of information in cloud games.

2.4.4 Game servers in the cloud

Cloud gaming, as described in section 2.4.3, means that the game clients run in the cloud. An alternative utilisation of cloud computing is using general-purpose public clouds to run traditional game servers, serving fully featured game clients. This configuration can be described as online games using cloud infrastructure. Time-dependent applications, like networked games, usually need to be custom-made, and can rarely be built upon typical web- or enterprise frameworks. Hence we need the kind of cloud service known as *Infrastructure as a Service* (IaaS)(Vaquero et al., 2009). This allows the developers to access complete virtual machines, with full access privileges, allowing them to run their fully customised programs in the cloud.

A game company planning to deploy an online game traditionally face huge costs for data centre space and bandwidth capacity. Because it is hard to predict a game's popularity before it is launched, the game provider is prone to choose over-provisioning in order to meet potential high demands. Over-provisioning boosts the costs of deployment further. Cloud computing is a popular way for other application areas to address this problem. Here, the customer is billed based on the resources used, and can scale resources dynamically. Game companies have, however, been reluctant to place their game servers in a virtualised cloud environment. This reluctance is mainly due to the highly time-dependent nature of games. Since providers must allow for normal network latencies in addition to processing time, any significant proportion of this delay incurred by overhead due to the cloud infrastructure should be noted.

2.5 Types of delay

While local games and cloud games use very different infrastructure, from the point of view of the user, delay works exactly the same way in both cases. All information follows the same path, and all delays work the same. We term this delay *interface delay*. For client-server games on the other hand, the situation is more complicated. Because of client-side prediction, some results are presented to the user after going through only the steps described in section 2.4.1, while other results have to go through the network and a server as described in section 2.4.3. In client server games, this extra delay is the sum of *network latency* and *server processing delay*.

2.6 Virtual reality

Regardless of the game genre or the delivery method, games are played on some sort of screen. Mostly the screen is something standing in front of the user. An alternative that has been worked on intermittently for many years is to attach the screen to the user's head. Combined with a system to track head movements, this can potentially allow for a much more immersive virtual world. If the user sees only computer generated content, such systems are called *Virtual Reality* (VR) systems; if the user sees virtual content superimposed on the real world, the system is called *Augmented Reality* (AR). Providing a stereoscopic view that moves with their head gives users an unprecedented visual immersion in the computer generated world.

The last years have seen a great increase in interest and popularity of VR solutions. Some are dedicated hardware made specifically for VR, such as Oculus Rift. Simultaneously, manufacturers of smartphones noticed that their hardware had all important components of a virtual reality display, such as a screen, motion and rotation sensors as well as decent 3D graphics rendering capabilities (Olson et al., 2011). Thus, they released *mobile VR solutions* which are essentially smartphones strapped to a headset. Among these are Samsung's Gear VR, HTC and Valves' new Steam VR and even the simplified solution Google Cardboard.

However, these systems all have the same potential problems when it comes to motion sickness and discomfort when using the VR solutions. Davis et al. (2014) investigated a condition they term *cybersickness* in analogy to motion sickness. They consider a range of options for the cause of these problems, delay among them. However, they do not quantify delay that might lead to symptoms. Latency, delay and frame-rate are only some of the issues that might create discomfort and motion sickness while using VR systems. Other potential sources of sickness are beyond the scope of this thesis. These include technical aspects of VR content such as lighting 3D scenes, camera animations, acceleration, position tracking accuracy and more.

2.7 Summary

Games are diverse, and commonly grouped into genres. We assume different genres, and even different games within the same genre, have different requirements for delay. Further, the technical architecture of the game might influence delay requirements because processing on the client can give responses faster than server processing. A local game only has local delay, while a remote game exhibits both local as well as network and server delay. We want to measure

delays in existing systems, both using mouse input and virtual reality systems.

Because games often mimic physical interactions, and always are interactive, we also focus on establishing empirical values for how fast a response is required for an interaction to be perceived as instantaneous. This does not set a firm requirement on how fast responses are required, but is a relevant design goal for most games.

Technically, two trends are also apparent. First, computers are becoming increasingly parallel, and second, many services are moving into the cloud. Thus, we want to create new game architectures that utilise these trends.

Chapter 3

Measuring delay in games

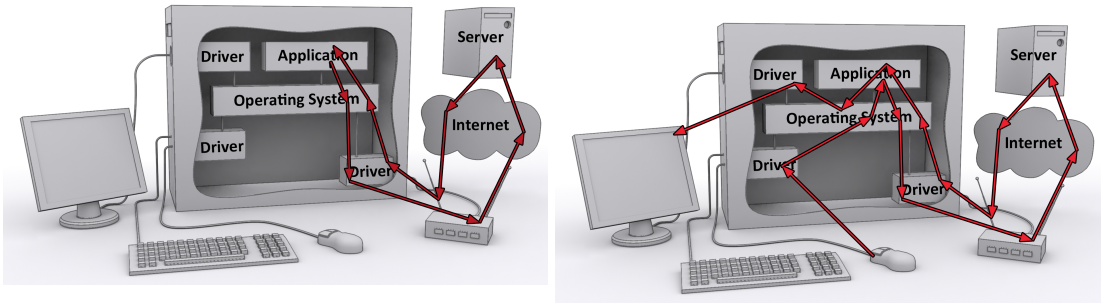
While developing games and other interactive applications, it is common to measure delays. Mostly, applications are instrumented to report various performance metrics, including response time. However, these measurements cannot capture the full delay. An application knows about input when it has received an input event, and considers the response completed when it has sent the result to the operating system. However, there are other sources of delay. Hardware, operating systems and drivers all may spend some time passing an event from the user to the application and again spend time passing the results from the application through the output device to the user.

Thus, to measure total delay in an interactive system, we could theoretically instrument every element between user interface input and the experienced result, which is near-impossible. As an alternative, we will employ *external* measurements. In our research, we look at two situations where local delay is important. One is a traditional user interface, where the users push a button and results appear on screen. The second scenario is virtual reality. In this scenario, the users turns their heads, and the scene on the display follows.

This chapter aims to answer question 1 of section 1.2: "How much delay is there currently in games?" and is based on work presented in "How much delay is there really in current games?" (Paper II, p. 121) and "Measuring latency in virtual reality systems" (Paper III, p. 127).

3.1 Related work

A surprising amount of research exists on the effects of delay in games that does not report actual delay from the time when the user pushes a button until the results appear on screen (Claypool and Claypool, 2006; Pantel and Wolf, 2002; Beigbeder et al., 2004; Claypool, 2005; Claypool and Claypool, 2010; Nichols and Claypool, 2004; Quax et al., 2004; Chen et al., 2006; Quax et al., 2004; Armitage, 2003; Claypool and Finkel, 2014; Amin et al., 2013). All reported numbers are about *added* delay, usually delay added by networks, but sometimes also delay added by processing on the server, as illustrated in 3.1a. Delays that occur locally on the client are, however, consistently ignored. To make the following discussion clearer, we need some terminology. *Total delay* refers to the duration from a user sends input to the results are available to the users. In a networked game this includes all the steps shown in figure 3.1b. Because the game itself does not have full control of input and output devices, this delay must



(a) Added delay, the fraction of delay reoported in most literature.

(b) Total delay for networked games.

Figure 3.1: Delay added by network as part of the full chain of delay.

be measured by equipment *external* to the system running the game. *Network delay* refers to the duration from an when event is sent from the local machine, to the server and back. This is distinct from network latency, because it includes processing time on the server. *Local delay* is the total delay minus the network delay and server delay. In local games, local delay and total delay are equivalent.

3.1.1 Measuring local delay

Previous work on button clicks and response time has used a consumer camera, which was configured to run at high frame capture speed, and film button clicks along with their results on screen. This is an imprecise and labour-intensive approach, because it relies on manually counting frames from seeing the button pushed until the results appear on screen. It is also limited in resolution to the frame-rate of the camera. However, it does give reasonable estimates for local delay. Renderingpipeline (2013) finds values between 76 ms and 112 ms for the game *Half Life 2* depending on how the game is configured. BlurBusters (2014) tests multiple games, getting results of 72–77 ms for *Battlefield 4*, 52–59 ms for *Crysis 3* and 22–40 ms for *Counter Strike: Global Offense*. Note that these results are from online sources. To the best of our knowledge, local delay after button clicks has not been described in recent, peer-reviewed work. These sources do, however, thoroughly describe their methodology and procedures, and the numbers are averaged across repeated measurements. This indicates that local delay constitutes a significant part of the total delay, at least in some hardware and software configurations. Therefore, knowledge about both network and local delays are important for a proper analysis of the effect of one of the components of user experience.

These numbers indicate that different games handle input in very different ways. Also, we know that network delays as low as 50 ms may affect player performance in some games (Dick et al., 2005; Jota et al., 2013). It is therefore important to investigate how player performance is affected by local delays of similar magnitudes.

The virtual reality scenario is better studied. *Virtual reality* involves head-mounted displays and motion trackers and allows users to rotate their head to see different parts of a virtual reality scene. These interactions are particularly latency sensitive, and various approaches have been suggested for measuring latency in such setups. Virtual reality was first studied for military applications, particularly flight simulators. Platt (1990) describes a full setup using head

mounted displays available at the time, and claims they were fast enough that delays were only noticeable during fast head movements. However, he does not report measurements on how long the delay actually is. Civilian work has used cameras (Kijima and Ojika, 2002) or light sensors (Swindells et al., 2000). What they all have in common is that they rely on a continuous, smooth movement of the tested devices. Di Luca (2010) summarised a series of them, and suggested a new approach, the most elegant we have found so far. Using a screen with a brightness gradient in front of the virtual reality headset displaying a duplicated virtual scene, he rotated the headset back and forth in a continuous movement. He then compared the phase difference of brightness values from light sensors on the display and in front of the headset. This setup seems to work well, though it is quite complicated to replicate. In addition, it is only able to measure delay in continuous movement. Further, this paper is some years old and hardware has advanced significantly since its publication.

3.1.2 Measured delay in cloud gaming

With the rise of cloud gaming, the awareness of total delay has increased somewhat. Chen et al. (2011) studied a cloud gaming system. They described the components of delay and present ways to measure each component. The components they describe are the following:

- "Network delay (ND): the time required to deliver a players command to the server and return a game screen to the client. It is usually referred to as the network round-trip time (RTT)."

- "Processing delay (PD): the difference between the time the server receives a players command (from the client) and the time it responds with a corresponding frame after processing the command."

- "Playout delay (OD): the difference between the time the client receives the encoded form of a frame and the time the frame is decoded and presented on the screen."

Chen et al. measured network delay using ICMP pings. The other components are measured using software hooks that are triggered by input and output events. Using this setup, they find that different cloud gaming providers have highly variable processing delays from 135 ms for the best cases to 500 ms for the worst cases. These numbers do not include network latency. This work suffers from one of the same limitations as previously mentioned in section 3.1: Ignoring delays produced outside the client application, such as drivers and screens.

3.2 Important components in local delay

Even in a local game without networking, some delay is unavoidable. This section discusses the parts of the pipeline that add most to the response delay in the local system. Components in a system, from user input to screen responses are to a large degree *black boxes*; documentation about how they work is often lacking, and details are considered trade secrets. Thus, the only way to evaluate these delays is by measuring.

Table 3.1: Some monitors and their response time.

Brand	Type	Response time
Apple	Thunderbolt Display	12 ms (Apple, 2015)
Dell	UltraSharp 24	8 ms (Dell, 2015)
Asus	ROG SWIFT PG278Q	1 ms (Asus, 2015)

3.2.1 Display devices

Display devices add delay, some of this delay is mentioned in the devices' specifications, but not all. Various types of display devices are in use. Monitors are the most common type of display device. These are specifically designed to be connected to interactive computers; some are even designed for gaming purposes with a focus on low delay. Other players use TV-screens or projectors, which are generally designed for movie and TV content. Virtual reality systems use either dedicated virtual reality displays or smartphones attached to the head using a simple rig. The reminder of this section discusses the different contributions to delay by display devices.

Update rate

Display devices receive and display updated images at a fixed rate. This rate is termed *screen refresh* rate. Most modern screens update at 60 frames per second (FPS), or every 16.7 ms. Some screens specialised for gaming or other response critical applications can update much faster, currently up to about 140Hz (BlurBusters, 2015). A new frame is read from a buffer and appears on screen after at least one such interval.

Response time

Response time denotes the time display panels take to change from one shade of grey to another, after the update has been sent from the computer. The exact procedure and colour values for this test are not standardised, and it is reasonable to assume that manufacturers choose the conditions most favourable to their product. Monitors vary wildly in this metric, from 1 ms for screens designed for gaming, to 12 ms for typical office screens, see table 3.1 for some examples. For TVs and projectors these values are generally not mentioned in the specifications. Because these devices are not designed for time-critical content, we can assume that numbers are towards the slower end of the range found for computer monitors. However, some modern TVs support game settings designed to lower delay.

Buffering and processing in display devices

Gaming consoles such as PlayStation and Xbox are usually connected to TV-screens or projectors rather than purpose-built monitors. These are not designed for time-critical content, but for high visual quality and the ability to adapt to a wide range of encodings, environmental conditions and user preferences. Thus, they perform internal processing, which requires internal buffering. Such processing and associated buffering adds delay, however we found no mention of such delays in manufacturer specifications.

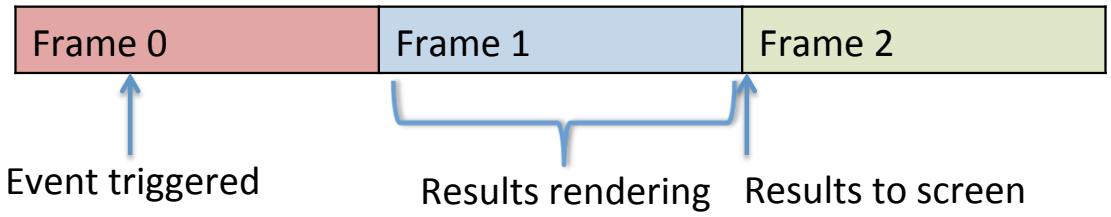


Figure 3.2: Timeline for events with double buffering.

3.2.2 Graphics hardware

A *frame buffer* is memory used for holding a rendered frame about to be displayed on the monitor. Modern games use at least two frame buffers. To avoid showing unfinished frames on the screen, drawing happens to one buffer while the other is being displayed. This practice is termed *double buffering*. Further, to avoid showing parts of two different buffers, it is common to wait for the next screen refresh before swapping. The terms *vertical synchronisation* or *vsync* are used, because historically the swap was performed when the electron beam was returned to the start of the frame, a period of time called the *vertical blanking interval*. Disabling vsync is possible, but introduces an artefact called *tearing* when the camera pans or objects in the scene move vertically. By showing the upper part of one frame and the lower part of the next, the boundary between the frames is visible as a horizontal line through the picture.

When double buffering is used, rendering follows the sequence, as shown in figure 3.2: Assume frame 0 is the frame during which an event from an input device is registered. Frame 1 contains the result of the event, and at the time of frame 2 the result is sent to screen. This gives a minimum of 1 full frame time from input event to result on screen. At 60 FPS this adds up to a minimum of one frame delay (17 ms) to a maximum of two frames (33 ms) delay. Many games have a target frame-rate of only 30 FPS. At this rate, the delay from screen refresh and frame buffer pipeline is 33 - 67 ms. Further, not all hardware is capable of keeping up with the target frame-rate at all times. Slow hardware leads to significantly longer delays.

An increased number of frame buffers in the pipeline increases this delay, because more steps are added between rendering and displaying data. High system load from the game itself or external tasks can lead to lower frame-rate, and thus add to this number.

3.2.3 Input device latency

The best gaming equipment has tailor-made drivers that have been tweaked for low latency performance. A good gaming mouse may add $\sim$2 ms (Renderingpipeline, 2013) to the latency. Devices that are not made for gaming may add more, but manufacturers do not document delay for such products. Delays from input devices are not only introduced in the device itself; drivers, operating systems — particularly time-slicing — and the applications themselves may add extra delay to the processing of an external event.

Dedicated virtual reality equipment has tailor-made hardware and drivers that have been

tweaked for low latency performance. The gyros of smartphones, on the other hand, are mainly intended for navigation or keeping the screen rotated the correct way, neither of which require fast response time or high accuracy, making these somewhat unreliable for use in virtual reality applications.

3.3 Experiments

To measure total delay from user input and screen output, we set up two experiments. Measuring delays between input and output accurately requires purpose-built setups. We run two different experiments, one measuring response time after a mouse click, and another measuring response time after an abrupt movement of a virtual reality device. These two experiments use the same system to capture output and measure durations, but require different equipment to capture input.

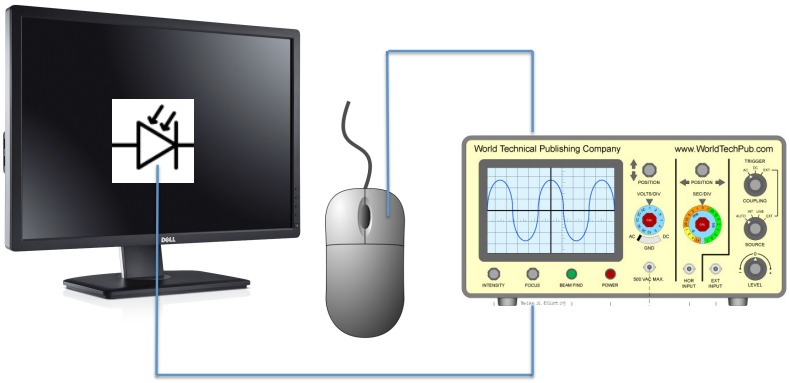
3.3.1 Mouse click

We designed a setup to measure the delay after a mouse click independent of the hardware and software used to play the game. This allows us to test real, commercial games on several different computers and screens, without modifying software or hardware. Results represent the sum of all delays present in the game. Delay from the input device, the game software, graphics card drivers, the graphics card itself or the screen are all added up along with any unknown sources of delay in the pipeline. Investigating how much delay each part of the test system introduces requires modifying the test system itself, locking the experiment to one pre-modified testbed rig. We want to be able to test the delay of a wide range of setups. Thus, a requirement to modify each system would defeat the purpose.

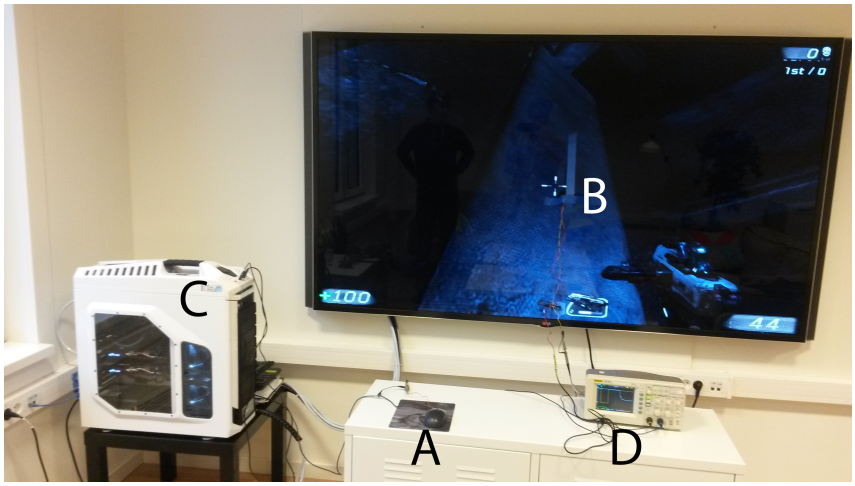
Previously, only hardware benchmarking sites and blogs have studied this issue, as described in section 3.1. BlurBusters (2014); Renderingpipeline (2013) measured delay by filming the screen and a button with a high-speed camera while pushing the button repeatedly. This approach has some limitations. First, it is limited to the capture rate of the camera. Secondly, it is difficult to judge exactly when the button was pushed, and lastly, it requires tedious manual work to analyse the videos.

To get more exact results, we used an oscilloscope to measure the timing difference in a setup shown as shown in figure 3.3b. We soldered a wire to the left mouse button switch, giving us direct access to the output from the physical device, bypassing any processing. This wire was connected to one channel of the oscilloscope. To measure the output, we used a sensor measuring light from the screen. Output from the light sensor would be proportional to the light emitted by the screen. We measured the internal delay in this setup by connecting one channel of the oscilloscope to a butten which simultaneously triggered an LED, which activated the light sensor connected to the other channel. This showed internal delays multiple orders of magnitude lower than those we are interested in.

To be able to measure a specific combination of computer, screen and game, we found a dark place in the game. There, we triggered an in-game effect that lit up the screen fast. It is important to keep in mind that not all actions in-game are meant to be immediate, but firing a weapon will in most games be. Many actions in-game use what in the art of animation is called



(a) Sketch of the setup used to measure delay in games.



(b) Photograph of the setup in practice. The mouse (A) triggers a shooting action in the game. The photosensor on the screen (B) captures the flash from the shooting effect. The PC (C) renders the game while the oscilloscope captures the events and measures the time between them.

Figure 3.3: The setup used to measure delay in games, shown schematically and as a photograph.

anticipation, meaning that part of the animation happens before the actual effect triggers. We wanted to measure delays added by technological limitations, not those that are deliberately added by designers.

We could thus measure and log the delay between the click of the mouse button and the resulting brightening of the screen, by comparing the flank representing the mouse click (yellow on figure 3.4) with the flank representing change in light from the screen (blue on figure 3.4). Results obtained using this setup are reported in section 3.4.1.

3.3.2 Virtual reality headset rotation

As mentioned in section 2.6, virtual reality systems are rising in popularity as devices to interact with games. Manufacturers are marketing both dedicated systems and simple rigs that let users convert their smartphones to virtual reality displays. As with traditional input methods, response time is an important metric in the performance of these displays. Hence, we extended our

Figure 3.4: Capture of delayed event. Yellow line represent output from the mouse, blue line represents output from the light sensor. Both are active low. The difference in time between the two flanks is shown in green in the bottom-left corner of the oscilloscope screen.

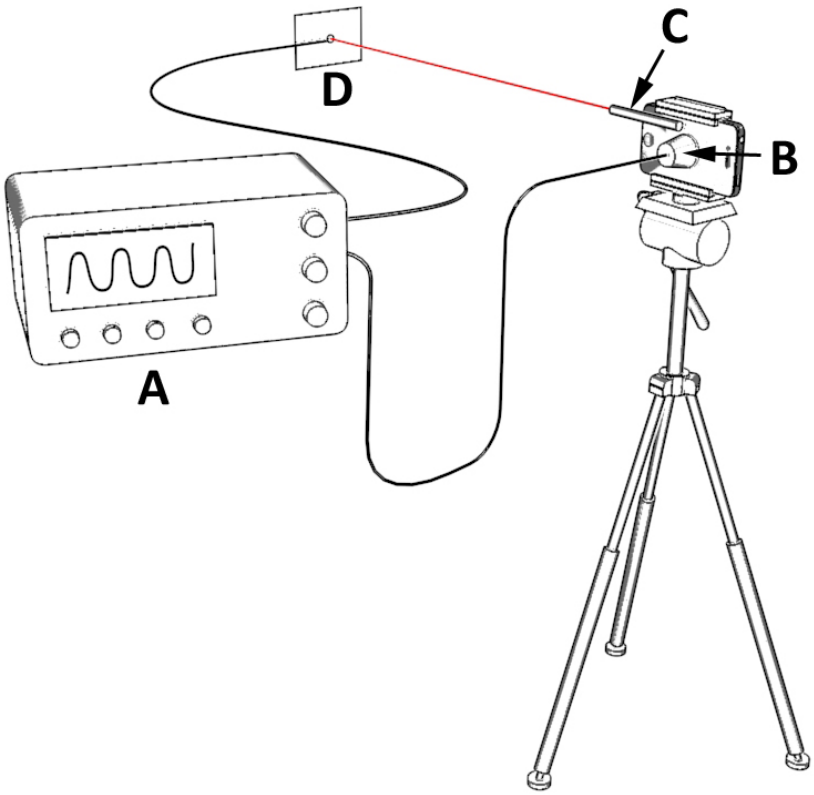
experiment to measure response time in such systems.

Virtual reality systems may utilise motion prediction algorithms, and games contain many abrupt movements. Thus, experiments measuring delay only in predictable movements will not give the full picture. For the virtual reality headset, we are interested in the elapsed time from the rotation of the headset until the resulting change is visible on the display. We want to look at response time for abrupt rotations, in contrast to previous work, which has looked at continuous predictable movement patterns.

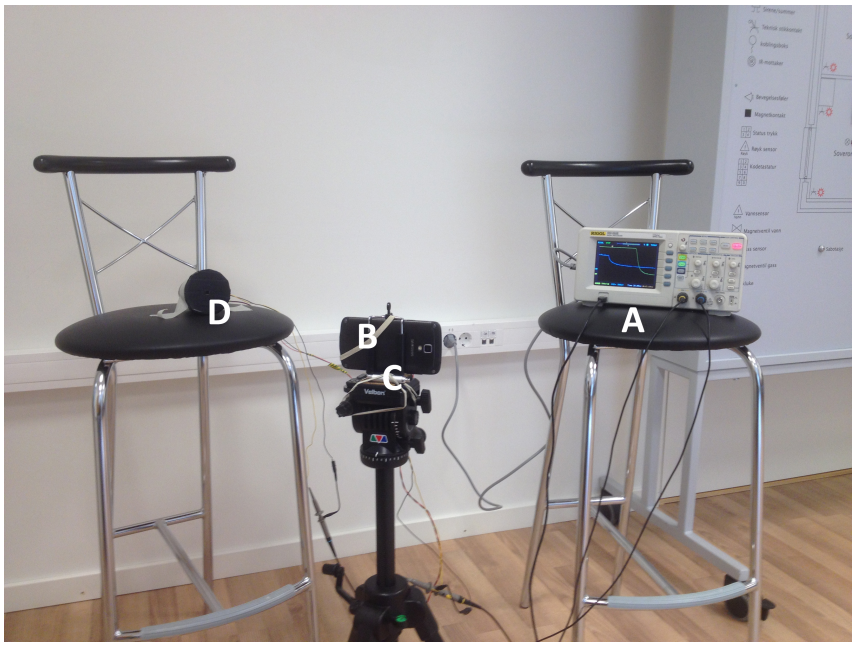
To measure response time after abrupt movements, we need the virtual reality device to run a program that detects a small rotation and as fast as possible change the displayed picture. A simple program that detects rotation and changes the displayed picture would solve this problem. However, we are interested in delays from actual 3D virtual reality software. Therefore, we used a popular game engine, Unity 3D[1], and created a scene that creates abrupt changes based on headset movement.

The virtual reality setup works with any virtual reality device. However it requires purpose-made software, because it is very difficult to set up a real game so that the screen is consistently black while even a small movement of the display device will turn it white or vice versa.

We tested multiple virtual reality devices. Oculus Rift is a dedicated virtual reality display solution, designed for this purpose. The other systems are smartphones, which developers have discovered have all the required hardware to run virtual reality application and at the same time function as headsets.



(a) Schematic drawing



(b) Photograph

Figure 3.5: The Physical setup with the Oscilloscope (A), light sensor (B) attached to Virtual device, Laser pen (C) and light sensor (D) picking up the laser.

Physical setup

The physical setup consists of the virtual reality device (Oculus Rift, Smartphone etc.) mounted on a camera tripod. One light sensor is attached to the screen of the virtual reality device to register the virtual scene shifting from white to black. A laser pen is also attached to the virtual device pointing at another light sensor approximately one meter from the tripod setup. Both light sensors are connected to an oscilloscope. When we move the virtual reality device by turning the tripod, the light sensor illuminated by the laser pen registers the disappearance of the light. When the movement is detected, the light sensor connected to the virtual device screen measures the light shift from the white plane disappearing in the virtual scene.

Virtual setup

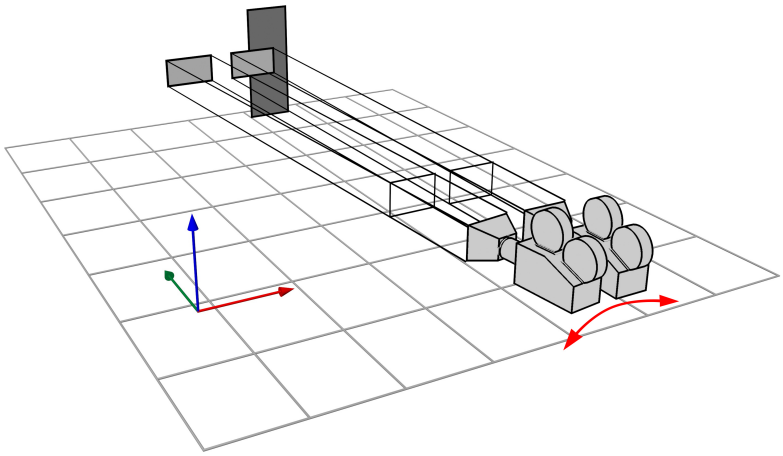


Figure 3.6: The virtual scene setup. The camera is zoomed far in and looking at a black plane against a white background far away. This geometry means that a very small movement of the head mounted display will change the scene from black to white.

The virtual setup running on the virtual reality device is a simple completely black virtual 3D scene with a white self-illuminated plane and a virtual stereo camera, as shown in figure 3.6. The virtual camera is set up with normal virtual reality movement controllers. For the mobile phone setup, the Durovis Dive SDK[2] was used while for the Oculus rift setup the Oculus Rift SDK[3] was used. We set up a virtual reality scene with an absolute minimum of content optimised for the highest possible frame-rate, giving consistent values around 3000 frames per second and to use the build in VR movement controllers. The OculusRift devices were connected to a fast laptop[4].

The scene shown in figure 3.6 consists of a white plane on a black background. The camera faces the white plane from a large distance and is set to a very narrow field of view. The scene

[1] `http://unity3d.com/`
[2] `http://www.durovis.com/sdk.html`
[3] `https://developer.oculus.com/`
[4] Windows 7, Intel i7 - 3740 QM CPU @ 2,70 ghz, NVIDIA Quadro K2000M - 2048 mb DDR3

Table 3.2: Results from mouse click delay measurements. Details of the various systems are in text and footnotes.

Experiment	System	OS	Avg.	Min.	Max.
UT3, default, vsync on	MacBook Pro	Win 7	95 ms	79 ms	102 ms
UT3, default, vsync off	MacBook Pro	Win 7	58 ms	52 ms	66 ms
UT3, optimised, vsync off	MacBook Pro	Win 7	33 ms	23 ms	38 ms
UT3, default, vsync on	Gaming PC & TV	Win 7	172 ms	167 ms	181 ms
UT3, default, vsync off	Gaming PC & TV	Win 7	128 ms	120 ms	138 ms
UT3, optimised, vsync off	Gaming PC & TV	Win 7	103 ms	93 ms	113 ms
UT3, default, vsync on	Gaming PC & Monitor	Win 7	112 ms	85 ms	127 ms
UT3, default, vsync off	Gaming PC & Monitor	Win 7	48 ms	35 ms	54 ms

is tweaked so that the display is completely white initially. Because of a narrow field of view, the camera is very sensitive to rotation. This means that even a small rotation in the headset leads to the screen changing colour from white to black, a change picked up by the light sensor.

3.4 Results

We report results from the two experiments are separately before discussing and comparing them.

3.4.1 Mouse click

Using the setup mentioned above, we first measured the delay of a small test application, before moving to real games. For each condition we repeated the test 10 times, reporting minimum, maximum and average results. We have tested one laptop and a PC with two different displays. The two systems were a MacBook Pro[5] and a powerful gaming PC connected to a large TV[6] screen. We also connected the same gaming PC to a monitor[7]. We ran the game *Unreal Tournament 3* (UT3) and timed a basic pistol shot, an event that is supposed to be *immediate* and produces a muzzle flash effect. For this game, we tested using different conditions. Firstly, we tested default settings. These had vsync off and a resolution of 1024 by 768. At these settings, the game ran at about 60 FPS. Then, we turned on vertical synchronisation and ran the experiment again. Lastly, we ran without vsync and with all settings optimised for fastest possible frame-rate. These kinds of tweaks are popular among professional gamers. With these settings frame-rates are highly variable, averaging around 300 FPS on the MacBook and 1000 FPS on the gaming PC.

As we see in figure 3.7 and detailed in table 3.2, response time is highly variable, even in the same game running on the same setup. Average delays in UT3 on the MacBook vary from 33 ms to 95 ms depending on the settings. The gaming PC and TV combination gave delays

[5]Retina, 15-inch, Early 2013, 2.4 GHz Intel Core i7, 16 GB 1600 MHz DDR3, NVIDIA GeForce GT 650M

[6]LG 84 inch TV model no. 84UB980V-ZA, 4.0 GHz Intel Core i7-4790K, 16 GB 1600 MHz DDR3, 2 x NVIDIA GeForce GTX 980

[7]Dell E248WFP

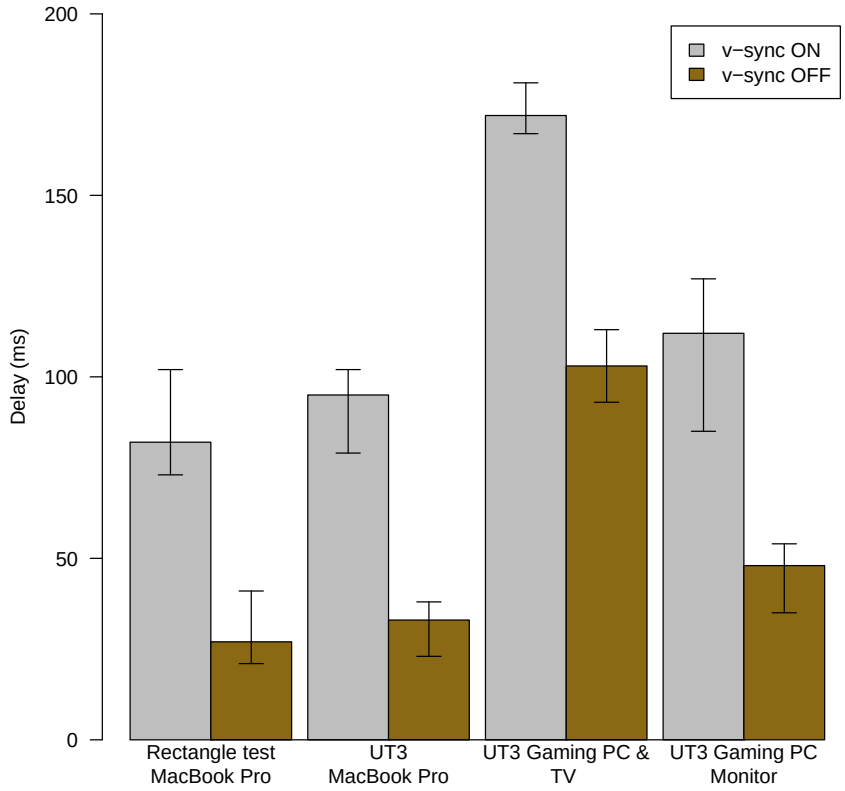


Figure 3.7: Average response time in the various mouse click configurations. Whiskers represent maximum and minimum measurements.

Table 3.3: Results from virtual reality headset movement delay measurements.

VR Display	Avg.	Min.	Max.
Oculus Rift dev kit 1, v-sync ON	63 ms	58 ms	70 ms
Oculus Rift dev kit 1, v-sync OFF	14 ms	2 ms	22 ms
Oculus Rift dev kit 2, v-sync ON	41 ms	35 ms	45 ms
Oculus Rift dev kit 2, v-sync OFF	4 ms	2 ms	5 ms
Samsung Galaxy S4(GT-I9505)	96 ms	75 ms	111 ms
Samsung Galaxy S5	46 ms	37 ms	54 ms
iPhone 5s	78 ms	59 ms	96 ms
iPhone 6	78 ms	65 ms	91 ms

between 103 ms and 181 ms, considerably slower. Connecting that same PC to a monitor gave better results, from 48 ms to 112 ms.

3.4.2 Virtual reality headset movement

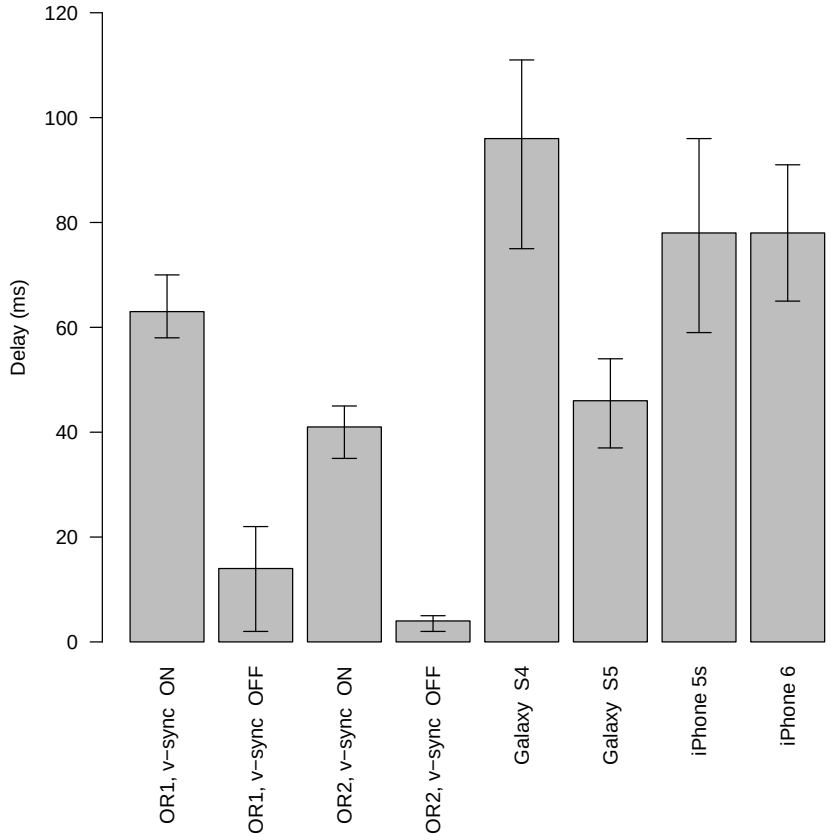


Figure 3.8: Response time in the various VR configurations. Whiskers represent maximum and minimum measurements.

Using the setup described, we measured the delay of multiple VR systems in two main categories, dedicated VR hardware and smartphones. Figure 3.8 shows the results from the systems we tested, detailed in table 3.3.

We tested two versions of OculusRift development kit, version 1[8] and version 2[9]. Regarding both these products, the developer provides sparse specifications, only claiming "a low-persistence OLED display and low-latency positional head tracking". For these, V-sync has a large effect on total delay. With V-sync off, the frame-rate in the application was as high as 3000 fps, while V-sync on forces the frame-rate to precisely 60 fps. Turning off this feature introduces visual artefacts, but reduces the delay significantly. With vertical synchronisation, Oculus Dev kit 1 had an average response time of 63 ms, while the newer Oculus Dev kit 2 averaged 41ms. Without vertical synchronisation, these dedicated virtual reality headsets can react very fast, Oculus Dev kit 1 averaging 14 ms while Oculus Dev kit 2 responded in 4 ms.

We also tested four different smartphones. These do not allow control of vertical synchronisation. They are much slower than the dedicated VR hardware, with delays close to 100 ms for most models. The exception is the Samsung S5, which has a delay less than 50ms. This result is similar to the Oculus Dev kit 2 with V-sync on. The screen in the Samsung S5 is actually the same as in the Oculus Dev kit 2, and the result confirms the similarity between the two devices and implies that the phones operate with V-sync on.

3.5 Discussion

The results from this research were quite surprising. Measured delays are considerably longer than can be expected from summing up reasonable estimates for delays described in section 3.2. Especially the setup using a TV as display is slow to respond. Why are these delays so long? Apart from the causes mentioned in section 3.2, it is difficult to draw clear conclusions. It seems that devices, such as graphics cards and even displays, buffer from one to several frames.

These results come with some uncertainties. Despite our efforts, the rendered scene might not turn instantly from white to black between one frame and the next. Intermediate frames should show up as plateaus in the oscilloscope output, but we cannot be completely sure that the animated movement did not pass through intermediate frames between white and black. However, consistent results indicate that this is unlikely. The measurement of the Oculus dev kit 2 without V-sync showed almost no measurable latency. Further, the Oculus Rift dev kit 2 screen flickered at 75Hz, creating a jagged curve on the oscilloscope display. This jaggedness made it difficult to measure the actual delay before the screen started to turn darker, hence our results from this equipment have an uncertainty of $\approx$5 ms.

For virtual reality, unsurprisingly, the dedicated hardware has much faster response rate. Further, the smartphones do not give developers access to control the vertical synchronisation setting, and it seems from the numbers that it is always on.

Because frame-rate is an easily accessible metric available in all games, this has become the focus of attention for anybody who are interested in how games perform. Consequently, players seek high frame-rate with more detailed graphics, and developers comply. The increased complexity in the hardware might have made additional buffers necessary, to merge data and synchronise different parts of the architecture. If a frame has to travel through multiple buffers the frame-rate must increase even further to get information through the pipeline. Total time

[8] `https://www1.oculus.com/order/dk1/`
[9] `https://www1.oculus.com/order/`

from input through the pipeline to screen is, as we have measured and reported here, likely more important to the users' experience than frame-rate, assuming the frame-rate is high enough for motion to appear fluid. Probable reasons for focusing on frames per second are a lack of awareness of the problem and difficulty in measuring total delay.

With the Oculus Rift we seem to see more focus on response time: This product is optimised for response time, and is the only system that allows delays to reach single-digit milliseconds.

Our observations in this chapter are relevant for interpreting the results in the research presented in section 3.1; none of those papers report or discuss local delay. It seems implicit form these papers that local delay is either insignificant compared to the delay values they are using. This chapter, conversely, shows that local delays are significant. For example, Amin et al. (2013) test perceived quality of experience in *Call of Duty: Modern Warfare 2*. They conclude that the experience is impaired for expert players by 100 ms network latency. However, the paper does not report how much local latency is present, neither details of the systems used in the experiment. If the local latency was 50 ms, the measured impairment would occur at 150 ms total latency. Conversely, if the local latency was 150 ms, the measured impairment would occur at 250 ms total latency. Because the paper does not report local latency, both of these options are equally likely. Further, there is no indication that network latency by itself is the cause of the measured impairments rather than total delay. Thus the results are difficult to interpret.

We have measured only a few systems for delay. This is enough to show the typical order of magnitude of delays as well as a range of possible values. The number of potential combinations of hardware and software is huge and new versions are released constantly. Thus, attempting to make a comprehensive list of systems and their delay is infeasible. Further, modern commercial software developers rarely allow users access to older versions of their products, making it impossible to recreate exactly an earlier experimental setup. Instead, we suggest research on delay in games measure and report the actual delay rather than going to great lengths to specify the details of the systems used.

3.6 Summary

In this chapter, we have described a system to measure response time between user input and screen response for two different kinds of input, a mouse click and rotation of a virtual reality headset. Further, we investigated response time in various systems. Results are highly variable depending on system. In many cases we found delays that are higher than expected from summing up known sources of delay. We see that the numbers are large enough to influence the conclusions of much previous research.

However, an important question is how users perceive the quality of the gaming experience. When is the delay between action and response noticeable, and when do users get annoyed? There exists work that partly answer these questions, but based on those experiments it is unclear if the conclusions are reliable. Next, we therefore perform thorough experiments to see where users can observe delay in such systems.

Chapter 4

Limits for observable delay

When working to improve time-sensitive interactive applications, it is not only important to know the actual delays and their sources, but also the amount of delay that is acceptable to the users for any given design. This criterion should be based on empirical, real world data. For any proposed architecture or optimisation, the criteria for success must be based on studies on how users react to variations in experience. A central question then is "How fast must an application respond to user input?" While this depends on many details of the application some general guidelines might be possible. By looking at this, more basic question, this chapter builds the foundation for our technological studies.

This chapter investigates how short delays between user input and system reaction must be to remain undetectable to humans. In other words, we attempt to empirically establish how fast reactions have to follow actions to be perceived as *instantaneous*, as defined in section 2.1. This might, however, not be sufficient; imperceptible delays might influence users subconsciously. If not handled correctly, delays and variations in delay might also affect fairness in a game. We explored subjective sensitivity to temporal delays between motor inputs and visual outputs in three repeated-measures experiments, using two slightly different methodologies.

This chapter aims to answer question 2 from section 1.2: "How fast should game systems respond to avoid deterioration of player experience?" The related work section of this chapter is based on sections from "Latency Thresholds for Usability in Games: A Survey" (Paper I, p. 107) and updated with newer sources. The rest of this chapter is based on "Can gamers detect cloud delay?" (Paper IV, p. 135) and "Instantaneous human-computer interactions: Button causes and screen effects" (Paper V, p. 141).

4.1 Related work

Data on acceptable latency in games, or on how player performance correlates with latency is relatively sparse in current literature. In some cases, multiple papers cite the same few sources of empirical data.

4.1.1 Controlled studies

Allowing participants to play games in laboratory settings with controlled latency is a natural approach to investigating the effects of latency. It gives the researcher control over all parame-

ters when the game is played. On the other hand, gathering datasets from a significant number of players requires significant time and effort.

Pantel and Wolf (2002) claim racing games to be the most sensitive class of game. Using a setup with two identical machines, with controlled latency between them, the authors run two very different experiments. First, they set up an identical starting position and perform identical actions on both sides, observing discrepancies. In the games studied, this leads to both players seeing themselves in the lead at the same time, even at the lowest tested latency of 100 ms. Next, they allow players to play actual games under varying latencies to the server. They find that the *average* player's performance deteriorates first, at their lowest tested latency of 50 ms. The beginners drive too slowly to notice this delay, while excellent players are able to compensate for more latency. Performances of the excellent drivers degrade sharply at 150 ms.

A much cited paper on this topic is Claypool and Claypool (2006). Most authors citing this paper simply use it as an explanation for setting acceptable limits to response times from the system they are evaluating. The value 100ms is frequently quoted.

However, Claypool's work and conclusions are much more nuanced, categorising games based on how they interact with the player as described in section 2.2. The paper describes a whole set of game types and situations and recommends latency limits. These limits are estimated at 100ms for *first person avatar* games, 500 ms for *third person avatar* games, and 1000 ms for *omnipresent* games. Further, the authors analyse different actions within each type of game. For each of these items, they have used empirical studies showing how player performance varies with network latency. The commonly quoted 100 ms figure is based on shooting accuracy in a first person avatar game, the most latency-sensitive class of games according to Claypool and Claypool (2006). It is important to notice that 100ms represents a point where player performance already has dropped off sharply from the previous data point 75 ms, shown in Claypool and Claypool (2006), figure 2. Because systems should be designed to withstand worst-case scenarios, the design goal should be below 75 ms. Claypool and Claypool (2006) is, however, mostly a secondary source, citing data from earlier work, which will be described in the next paragraphs.

The 100ms estimate for *first person avatar* is from Beigbeder et al. (2004). They set players in front of computers running the game *Unreal Tournament 2003*, and give them a set of tasks, such as moving in a specified pattern and shooting at moving targets. The widely quoted number seems to originate in an experiment where shooting precision was tested. The experiment was run three times by two different players at each latency level. Results from such an experiment are not sufficient to draw any clear conclusions. Other factors were also analysed in this work, but are of little statistical relevance due to the extremely low number of participants. Other numbers cited in Claypool and Claypool (2006) are similarly from studies using an extremely low number of participants.

For *omnipresent* games, the most relevant data comes from Claypool (2005), which in turn uses most of the data from Sheldon et al. (2003). The papers do not establish any clear threshold for latency in such games, simply concluding that 1000ms should be completely safe. Splitting the games into different types of interaction receives significant focus, but differences in skill levels of players are not evaluated. Only two players participate in either study. For the last category of game, third person avatar, the data comes from Fritsch et al. (2005), which evaluate the performance of two players playing the game *Everquest 2* under different conditions.

Claypool and Claypool (2010) further elaborate on the categories of games and actions. Each action is described by two parameters; deadline and precision. Deadline is the time an action takes to complete, and precision is the accuracy needed by the player.

To investigate how sensitivity to latency varies with these two parameters, the authors modified a game, *Battle Zone capture the flag (BZFlag)*, so these parameters could be controlled directly. Each scenario was then played out using computer controlled player avatars called *bots*. They do not clearly justify that bots are an accurate model for how human players react to latency; neither do they cite any research indicating that this is the case. The hypothesis of a correlation between each of the variables and latency sensitivity was supported, but not strongly.

Team sports games are played in a somewhat different manner to the others mentioned here. Usually you have control of one character at a time, as in a third person avatar game, but you switch character often, depending on who is most involved in the action, as if in an omnipresent game. Nichols and Claypool (2004) study the sports game *Madden NFL Football*, and conclude that latencies as high as 500 ms are not noticeable.

Quax et al. (2004), set up a 12-player match of *Unreal Tournament 2003* in a controlled environment. Each player is assigned a specific amount of latency and jitter for the duration of the match. After the match, the players answer a questionnaire about their experience in the game. This study still uses relatively few players, but they are able to conclude that 60ms of latency noticeably reduces both performance and experience of this game. They did not find any effect of jitter.

Amin et al. (2013) also run controlled experiments, but use subjective measures on the FPS game *Call of Duty Modern Warfare 2*. Further, they graded participants according to gaming experience. They conclude that the most experienced users are not satisfied with latencies above 100 ms.

To simplify the gaming scenario, Stuckel and Gutwin (2008) used a rudimentary cooperative game. In their study, they define the concept *tightly coupled interaction* as "shared work in which each person's actions immediately and continuously influences the actions of others". Such interactions are a cornerstone of games, whether co-operative or adversarial. Evaluating user performance under different delay situations, they conclude that local delay is actually preferable to delay hidden by the system, since it allows users to compensate.

In summary, controlled experiments show clearly that performance decreases as delay increases above a threshold. It is, however, not clear what this threshold is for all types of games. These papers also in general include relatively few subjects and repetitions.

4.1.2 Observational Studies

To avoid the limitations of controlled studies, others have taken different approaches to determining acceptable latency for games. To get around the primary limitation of controlled experiments, the resources and time required to gather data, some researchers choose to gather data from games as they are being played across the net. This gives access to large amounts of data, but the researcher must rely on random variation in conditions.

Henderson (2001) wrote the oldest paper we found on this topic, and it is still relevant. Running an FPS game-server open to the Internet, the authors observe player behaviour and how this is affected by network delay. Most players connecting to their server had network

latencies in the interval 50 ms to 300 ms. Beyond this, their only result relevant here is that players with delays over 400 ms seem much more likely to leave immediately.

Armitage (2003) set up two different servers for the game Quake 3 and monitored the latencies experienced by the players joining the servers. They assume that players only join servers that have acceptable latency. Henderson (2001) contest this assumption, finding no correlation between delay and players joining or leaving game servers in a similar game. This methodology should give insight as to how much latency players think is acceptable. However, the study does not take into account which other servers are available, so the results could be influenced by the presence of lower latency servers or lack thereof. Additionally, the latencies players think are acceptable might not be the same as the limit where their performance degrades.

Another interesting approach is that of Chen et al. (2006). They examine an online RPG, *ShenZhou Online*. By Claypool's classification, this game would be a third person avatar game, and hence be less sensitive to latency than the games discussed earlier. Instead of using a controlled lab environment, the authors chose to analyse network traces from an existing, running game. They asked the question: "Does network QoS influence the duration of play sessions?" (Chen et al. (2006)) The Quality of Service (QoS) factors they examined were packet loss, latency and jitter. Their hypothesis was that if the underlying network conditions affect the players negatively, it should show up in the players' enjoyment of the game, and hence their motivation to keep playing. Between 45 and 75 ms RTT, the authors find a linear correlation between increased latency and decreased game session length. For standard deviation of latency, a measure of jitter, the correlation is even stronger. A finding which contrasts directly with Quax et al. (2004), who find no effects of jitter. The most probable explanation for this is that jitter is handled better in that game. At extreme RTT values or extreme jitter, the trend is reversed though, and the authors surmise that there is a group of players who are used to bad connections, and another group of players who keep the game on while not really paying attention. These results indicate negative impact of latencies much lower than the 100ms mentioned in earlier literature. Session length as indicator for player satisfaction is an ingenious approach, but the chain of effect from network latency to session length is complicated, and there is significant room for hidden variables.

Dick et al. (2005) use two separate methods to investigate how latency affect players. Using International Telecommunication Union's *impairment scale* (International Telecommunication Union (ITU-T), 2014), which defines a rating called *Mean Opinion Score* (MOS) ratings, they ran an online survey asking people how much delay would cause each level of impairment. Players reported they could play *unimpaired* at up to about 80 ms, and *tolerable* at 120 ms for most games. Further, the authors ran a controlled experiment testing different latencies. Their results show large differences between games, with the most sensitive game *Need for Speed Underground 2* showing impairment even at the lowest tested delay of 50 ms.

In summary, while details vary, these observational studies draw similar conclusions to the controlled studies described in section 4.1.1. Latency affects gaming experience; but the exact values required for this effect is unclear.

4.1.3 Cloud Gaming

As described in section 2.4.3, cloud gaming changes how delays are exposed to players. So far, cloud gaming mostly eliminates the possibility of client side prediction and smoothing, thus presenting all delay directly to the users. Thus, this scenario must be studied separately from the ones discussed above.

Latency in cloud gaming is much less studied than for client-server games. Jarschel et al. (2011) test players' subjective experience of varying network latencies and amount of packet loss in cloud gaming. Using a setup where the gameplay was transferred over a network mimicking the cloud scenario, the authors introduced varying Quality of Service (QoS) parameters and asked their 48 participants how they liked the service in each scenario. They concluded that a latency of 80ms was noticeable in the fastest-paced game. In all games, packet loss in the stream from server to client was extremely detrimental to the experience. Because their setup used UDP/RTP for all communication, packet loss would result in reduced video quality. However, their conclusion can only be used to put an upper bound on latency sensitivity in cloud gaming, because they conducted no experiments between latencies of 0ms and 80ms.

Claypool and Finkel (2014) use both subjective and objective measures to investigate delay in cloud gaming. Testing one racing game and one puzzle game based on physics simulations, the authors induce varying degrees of delay to running game sessions. For subjective experience, the paper uses a self-reported Quality of Experience metric. As an objective measure the authors use actual player performance as measured by points in the games. They conclude that 100ms delay degrade performance up to 25%.

4.1.4 Results from psychophysics

The question about how much delay affects users in games is part of the larger question of how delay affects humans in general. The discipline of psychophysics studies human senses in general. One concept they study is *sensory threshold*, the magnitude that a signal needs to have to be perceived, which is analogous to the question of how long a delay needs to be before humans perceive it.

Humans are very adept at handling and acting on objects, facilitated by both the motoric and the visual systems, along with other inputs. *Sense of agency* refers to the experience of being the direct cause of an event, and this term encompasses the expected delays that follow many actions (Haggard and Chambon, 2012). Indeed, one study found that participants maintained the sense of agency from a joystick controlling the movements of an image for intervals as long as 700 ms (Ebert and Wegner, 2010). At the same time, these participants were clearly aware of the delays much lower than this, though the authors do not draw any conclusions about how short delays are noticeable.

This scale of delays can approximate those that follow real physical events, where consequences are stalled by the time taken to traverse a distance. However, many human-computer interactions involve series of inputs and outputs and these require far more speedy reactions. Whether typing in text, shooting at moving targets, or moving a cursor across the screen, most users expect instantaneous responses from the system. The higher demand for this type of human-computer interaction is emphasised by the findings of an experiment that compare the temporal boundaries for the sense of agency and the sense of simultaneity (Rohde et al., 2014).

Participants were asked to push a virtual button and watch for a visual flash, then make a judgement on the simultaneity of the events, or on the event serving as the agent. Ingeniously, the virtual button was implemented using a haptic feedback device that allowed tracking the participants' movements well before they pushed the button. Because movements leading up to a button-push are predictable, the researchers could trigger the flash *before* users believed they had pressed the button. On average, the button-push was perceived as the agent as long as the visual flash did not lag by more than ≈400 ms; conversely, the two events were judged as simultaneous, at greater than chance rates, when the flash delay stayed below ≈250 ms. Interestingly, events were also judged simultaneous when the flash happened up to 50 ms before the push.

Jota et al. (2013) examined users' sensitivity to delay in touch interfaces. They constructed a touch-sensitive screen with the ability to display extremely fast responses (down to 1 ms). Participants were asked to drag an object to a target using a touch gesture. The researchers used time taken to complete this task as a measure of user performance. Users were sensitive to temporal delays for actions that require pointing and dragging. The authors found significant decrease in task performance for delays delay down to ≈25 ms. Pointing and dragging in touch-sensitive displays introduces some caveats though. If the user drags an object at constant velocity across the screen, temporal delay appears to the user as spatial offset from the finger to the object, and it is possible that the measured effect is due to this spatial offset rather than the temporal delay.

Still, humans are capable of adapting to fairly long temporal delays (235 ms) between movements of a mouse and movements on a screen, although this becomes increasingly difficult as the visual task speeds up (Cunningham et al., 2001). Clearly, instantaneous and simultaneous are not synonymous with zero delay, a computational impossibility. Yet, these and similar human-computer interactions place strong demands for speedy responses on a system. Moreover, studies on multi-sensory and sensorimotor processes have demonstrated that the human perceptual system is adaptable and quite capable of compensating for short temporal offsets between corresponding signals (Rohde et al., 2014; Fujisaki and Nishida, 2009; Heron et al., 2009; Occelli et al., 2011).

These results are immediately relevant to real-time games; a delay that people notice in such simplified interactions would be detrimental to gaming experience. However, the research described in this section is based on interactions that do not map clearly to how players interact with games.

4.1.5 Limitations of existing work

Although we have found a substantial amount of research into how latency and delay affect interactions both in general and in games, not all questions have clear answers yet. Many interesting questions remain in this area. We chose to focus on the effects of delay in simplified interactions modelled after interactions in real games, while also gathering information on the participants' experience in playing games.

Unclear classification

The game classification described in section 2.2 does not reflect the full spectrum of games, as games of each type can have significantly different latency requirements. In the research summarised in this article, conclusions about noticeable latency in third person avatar games range from 45 ms (Chen et al., 2006) to over 500 ms (Claypool and Claypool, 2006). Different games in the same category have very different modes of interaction. Third person avatar games can have very different ways of interacting with the environment. Some are 3D graphical representations of very abstract game mechanics, where relative positions and timing are almost irrelevant, while others are highly detailed simulations of realistic physics where small changes in position and timing make for large changes in the outcome of actions.

For first person avatar games, the differences often lie in playing style. Some games focus on tactical movement and cover (for example *Metal Gear Solid*), while others rely purely on reaction time and precision motor skills (for example *Unreal Tournament*). It is likely that the last category has stricter latency requirement than the first. No studies found in this survey address these questions.

The distinction between these types of games is also blurred by the fact that many games allow multiple points of view. Role playing games and racing games commonly support both first person and third person view modes for the same game; leaving it up to the user to decide.

Even the status of omnipresent games is not entirely clear. Some are clearly slow-paced and highly strategic, but not all. The well-known game *Star Craft* is one of the most popular games in this category. The papers cited here all seem to agree that this means it should not have strict latency requirements. However, this game is played as a professional sport in parts of the world. In these matches players take around 400 individual actions per minute. This equates to 150 ms per action (Lewis et al., 2011). It would be interesting to investigate if players playing on this level require latencies at least lower than the time between actions. We have not had the opportunity to work on that in this thesis.

Participant and controlled environments

From all these studies there are no clear, consistent results available, and the diversity in game scenarios make comparisons challenging. Studies suffer from uncontrolled environments or very limited numbers of participants. Studies using both a controlled environment *and* a number of participants that is large enough to do statistical analysis would do a lot to clarify the situation, but we have not found any. Further, information on the background of the participants would help interpreting the results, yet is only mentioned in a few of the studies. While designing a game aiming at a specific audience, the designers should have data available that is relevant for that target audience. The results from psychophysics similarly employ experimental setups that are difficult to relate to games. We attempt to use the psychophysical approach using experimental setups that are closer to a gaming situation, and gather data on how much prior experience playing computer games participants have.

Correcting for local delay

Lacking in all publications referenced in sections 4.1.1, 4.1.2 and 4.1.3 is a consideration of the local system delay. All of these studies report network latency. The network latency studied may be from actual network conditions or artificially induced. What is missing from the latency analysis is reports on how much *local delay* is present in the test setup. Local delay describes the time from a users sends an input until the result is visible on screen, for actions that do not need to traverse a network. Discussions on the influence of local latency for the player tolerance and experience are also missing. Such local latency can constitute a large part of the total latency and vary considerably depending on the hardware used and software configurations. Therefore, knowing what share the network and local delays constitute is important for a proper analysis of the effect of one of the components on user experience. Our research measures and compensates for local delay.

4.1.6 Summary

We started this chapter by asking how much delay is acceptable to players. Despite much research into the topic, we have found no clear answers. Results vary based on experimental methodology. While we have numbers as low as 25 ms (Jota et al., 2013), we also have numbers up to 250 ms (Rohde et al., 2014). Different genres of games clearly require different response times, but the literature does not agree on which games are the most sensitive. For cloud gaming, the amount of research done so far is too limited to draw conclusions, but it seems numbers fall into the same range as for client-server games.

4.2 Method

We know from section 4.1 that there is a limit to how long delay we may have in a game before the quality of experience or the players' ability to succeed in the game deteriorates. Previous work is however not conclusive, with two clear limitations. Firstly, most of the previous research has focused on network latency, rather than local delay or total delay. Secondly, working on real games introduces many confounding variables that are difficult to correct for. Further, real games are unpredictable and need long experiment runs in order to give statistically valid results.

To extend the existing research, we decided to experiment on simplified interactions. Simple interactions have some advantages compared to studying real games. First of all, we can create a consistent environment for all iterations, making analysis simpler. Secondly, investigating simple interactions allows us to use many more iterations, giving the results more statistical weight. It also allows us to focus more closely on human capabilities, without considering the effects of game design and implementations.

Simplifying interactions in this way also has some disadvantages. The clearest disadvantage is that results become more difficult to relate to actual games. When presented with a simple experimental setup, a user may concentrate fully on the one interaction being studied, while a player of a game usually has to process a continuous stream of complex stimuli. Such distractions could lead to less sensitivity to delay. On the other hand, the entertainment, immersion and

competitiveness of a game could lead to deeper focus and thus make distractions such as delay more. Further, when not using an actual game, we can not directly measure how much delay affects game performance. Instead, we measure how much delay users can *notice*. A conclusion about how short delays participants notice in such an artificial setting might thus not apply to a any actual game, but does give indications that are relevant. Because the delays we add are not linked to any technical source, but introduced artificially, we use the term *motor-visual delays* for the delay between the input, that is motor, action and the visual output being produced on screen.

4.2.1 Experiment design

Figure 4.1: Experiment setup for investigating sensitivity to delay. The screen presents stimuli while the button receives input.

In order to investigate sensitivity to delay using simplified interactions, we designed two different experiments, or from the point of view of our participants, *tasks*. Both used the same setup, illustrated in figure 4.1. The first experiment we call the *adjustment task*, and the second experiment introduced a variation of the more common simultaneity judgement task (Rohde et al., 2014; Occelli et al., 2011; Fujisaki and Nishida, 2009) we call *discrimination task*. We ran the first experiment separately and published the results (paperIV) before continuing to the next two experiments (paper V). Both sessions happened in a computer lab at *Westerdals – Oslo School of Arts, Communication and Technology, Campus Galleriet*.

For the first experiment, we recruited 13 female and 28 male volunteers to participate in the experiment. They were aged between 19 and 43 years. We ran the next two experiments over one session. To minimise the potential learning effects across the experiments, we reversed the order of them after half the participants. For this experiment we recruited 10 female and 41 male participants, aged between 19 and 33 years. All experiments used *Griffin click+spin USB controllers*[1], to register participants' adjustments and responses. These are simple controllers

[1] `http:/store.griffintechnology.com/powermate`

often known as *jog-shuttles*, comprised of a big click-button that also serves a rotating wheel. The rest of the experimental equipment unfortunately had to be changed between the experiment runs, as the computers used for the first run were scrapped before we started on the second run of experiments.

In addition to the two experimental procedures, we designed two simple types of stimuli, a *static* type and a *rotating* type. These static visual stimuli consisted of black discs on a white background. When the user pushes the button, the disc either appears or disappears alternately. The delay of this change is the experimental variable. We included two disc sizes, with diameters of 20 and 200 pixels. At a viewing distance of 60 cm, these disc sizes correspond to visual angles of 5.6° and 0.56°, respectively.

The rotating stimuli were simply discs moving in continuous circles. The disc rotated either slowly or quickly (0.2 or 1 revolution/second), with the speed of rotation varying randomly from trial to trial; the initial direction of rotation was also randomised across trials. When the users pushed the button, the rotation would change direction after a delay, the experimental variable.

A questionnaire handed out prior to the experiment, assessed participants' gaming experience, specifically, how many years they had played and the average number of hours played during a week. Because music is a time critical activity somewhat related to playing computer games, we also asked about musical experience. For the experiment using moving stimuli, we also asked about a list of game genres.

The two different types stimuli, static and rotating, were combined with two different experimental procedures for a total of four combinations. Due to the evolving nature of our research we did not test one of them, the combination of static stimulus with the discrimination task.

4.2.2 Adjustment task

For the adjustment experiment, each trial commenced with an initial delay (100, 200, 300, or 400 ms). Participants were instructed to push the button on the jog-shuttle to trigger the visual stimulus and turn the wheel to adjust the delay between their push and the visual change. Due to the lack of reference points, participants were always unaware of the physical value of the delay. Values for delay could be adjusted from 0 to 500 ms; if adjusted past these extremes, the values would gradually decrease or increase away from the extreme. If a participant made an adjustment from 200 ms to 100 ms, another identical adjustment would set the delay to 0. From there, the next identical adjustment would take the delay back up to 100 ms. This ensured that participants could not simply rotate the wheel to the end-point to achieve zero delay.

Because the controller itself has no indication of how far you have moved it, or how much it has affected the delay, each attempt would essentially be a shot in the dark, and finding a spot where the participant does not notice the delay can take a very long time. Thus, a visual guideline in the form of a simplified clock face helped participants keep track of rotations made with the USB controller. We encouraged participants to use the clock face to keep track of their adjustments and the temporal granularity; one full rotation of the circle represented one full adjustment round, bringing participants back to the initial delay.

Participants were allowed to spend as long as they wanted on each trial, but for the moving stimuli they had to make a minimum of five button-clicks before proceeding to the next trial. Each trial therefore involved a series of wheel rotations and button-clicks before reaching the

point of no delay, in a sequence illustrated in figure 4.2. With four levels of initial delay, two levels of rotation speed and two levels of rotation direction, along with two repetitions of all conditions, the full adjustment experiment included 32 trials and took approximately 10 minutes to complete.

To get participants acquainted with the task, they were first presented with two practice trials, from which data was discarded. Then they were given the opportunity to ask questions, before they completed the full experiment at their own pace.

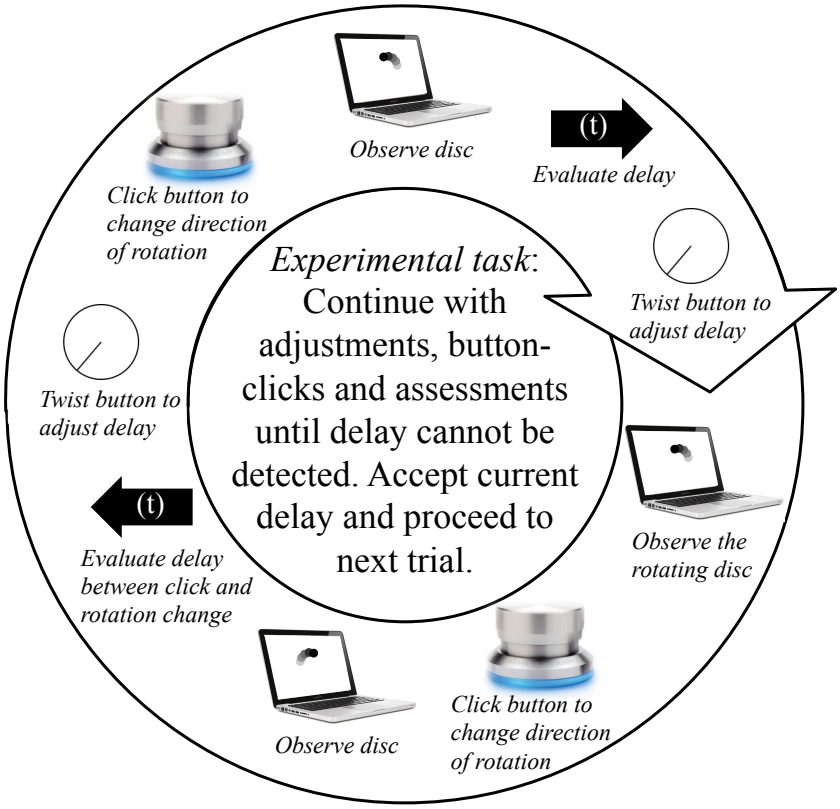


Figure 4.2: Circular timeline illustrating the experimental procedure for the adjustment experiment. The experiment starts with an initial push of the button and a delayed change in rotation direction and it continues for as long as it takes the participant to adjust the delay down to an imperceptible level.

4.2.3 Discrimination task

In the discrimination experiment, each trial involved a single button-click with a corresponding change in the direction of rotation. The delay between the click and the directional change varied randomly between 11 pre-established values (0, 20, 40, 60, 80, 100, 140, 180, 220, 260, and 300 ms). We asked participants to click the button and pay close attention to the visual change. They were thereafter prompted for a judgement on the simultaneity of the motor and the visual events. Participants provided their responses by turning the wheel left or right to choose either the *immediate* or the *delayed* response options. We included four repetitions of

all delay, direction and speed conditions, making a total of 176 experimental trials. Again we present the participants with practice trails and gave them the opportunity to ask questions. With these short trial presentations, the experiment duration was on average 12 minutes.

4.3 Limitations

No computer system can respond without delay. Any delay inherent in the experiment setup must be considered when reporting results from participants. Even when we attempt to introduce 0 ms delay, the system delay is present. Using the setup described in section 3.3.1, we measured the average system delay of the experiment setup to 51 ms.

In the adjustment experiments, when accepting a value, participants only know that they did not perceive any delay. Accepted values might not correspond to the exact minimum threshold for detection, rather any point below that threshold. Our collected data may therefore be uniformly distributed over a region of imperceptible delays. This concern introduces inaccuracies in the results. These are all considered when we interpret the data.

In the static stimulus adjustment experiment, we allowed participants to accept the current delay at any time, even before they had experienced the experiment. Therefore, any initial delay value that participants accepted without at least five adjustments was categorised as an accidental accept; accidental accepts were labelled as missing values and treated like such for the main analyses. For the moving stimulus adjustment experiment, we were aware of this problem and did not allow participants to accept any values before they had clicked at least five times.

Lastly, for the static stimulus experiment, the knobs adjusted delays in increments of 25 ms. This issue was fixed before the moving stimulus experiment, when the resolution of the knob was improved to 5 ms.

4.4 Results

Because we have results from the adjustment task from both types of stimuli, and these are directly comparable, we present these results side by side. First, we look at the visual factors in each study, then we compare the studies.

4.4.1 Results from adjustment task

In addition to the question of how much delay is noticeable, we also looked for variables that could influence the temporal sensitivity to motor-visual delays. We explored factors relating directly to the visual presentations. There is no reason to expect our data to follow the normal distribution; because values can not be lower than 0, we only have one tail. Therefore, we run non-parametric tests.

Static stimuli

We applied the *Wilcoxon signed rank-sum test* (Wilcoxon, 1946), a non-parametric hypothesis test, when we compared the related samples from a variable with two levels, and the *Friedman*

test (Friedman, 1937), a similar multi-variable test, for variables with three levels. The initially presented delay between the first button-click and disc-flash varied randomly between 200, 300, and 400 ms. However, when running a Friedman test, we found no indications that the initial delay influenced the final accepted delay ($\chi^2(2) = 0.79, p > .6$). Furthermore, the flashing disc also varied randomly in size. Still, the Wilcoxon test revealed no significant difference between the 20 pixel and the 200 pixel discs ($W = 660, p > .5$). Because none of these factors were significant, we collapsed the data and established a median of 36 ms for adjusted delays.

Dynamic stimuli

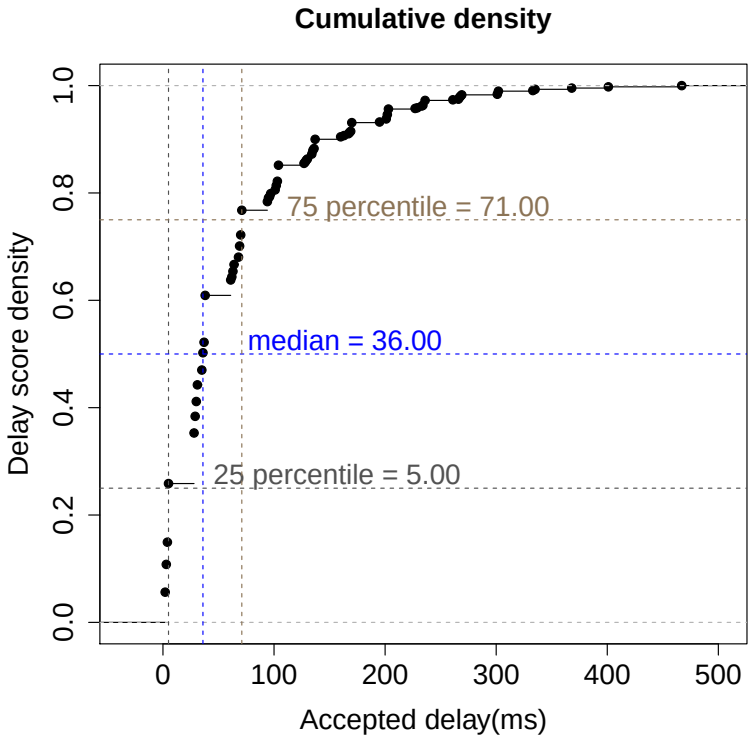
We ran two Wilcoxon signed rank-sum tests to investigate potential variations in delay sensitivity between the two initial disc rotation directions ($W = 338137.5, p > .5$) and the two disc rotation speeds ($W = 326454, p > .4$), we also ran a Friedman test to explore differences due to the initial delay values ($\chi^2(2) = 2.348, p > .5$). None of the tests revealed significant differences between the conditions. Following this, we collapsed scores across presentation modes and established the overall median for adjusted delays to 40 ms.

To establish detection thresholds for motor-visual temporal delays, we plotted an empirical cumulative density distribution with all delay scores. From this distribution, we derived the 25$^{\text{th}}$, 50$^{\text{th}}$, and 75$^{\text{th}}$ percentiles. The distributions and percentiles are portrayed in figure 4.3. When interpreting these results, keep in mind the limitations outlined in section 4.3, primarily local delay and the possibility of accepting delays well below what participants can detect. Also note that the steps of figure 4.3a are an artefact of the limited resolution available for adjustment in this experiment.

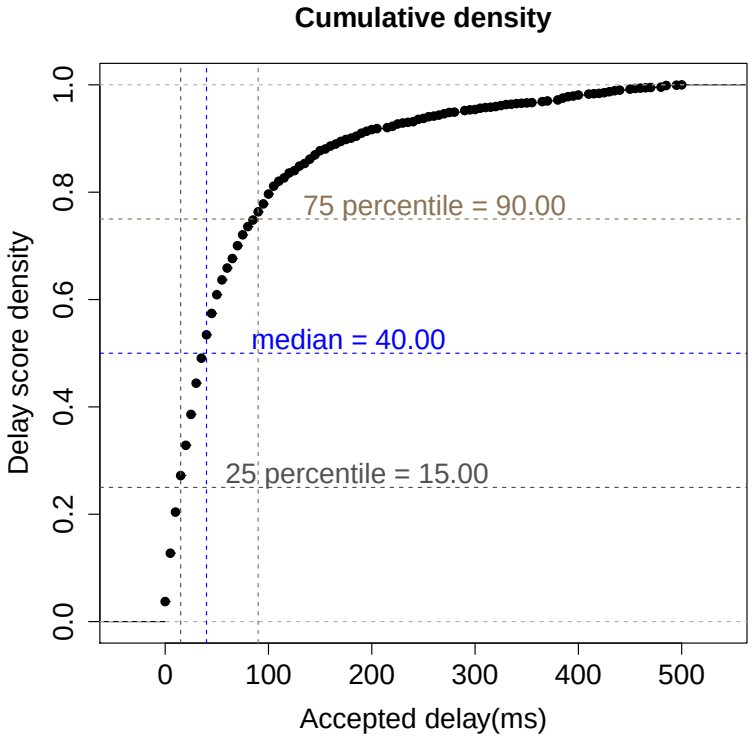
For a closer look at these distributions we present density plots in figure 4.4 of each participant's median accepted value. From these distributions, we found that the mode falls very close to 32 ms for both experiments, a lower value than the median. Furthermore, we observed an asymmetrical distribution, where the majority of scores centred on the mode, but a long tail of delay scores extended close to 400 ms. The dynamic stimulus produces a longer tail.

Participants were encouraged to spend as much time as they needed on the experimental task. Because participants can gradually work towards lower delay in this experiment, using more time and effort on the task may allow them to get closer to their sensory threshold. We observed great variations in task effort between individuals. During each trial, participants adjusted and tested the motor-visual delay by turning and clicking the knob, as outlined in figure 4.2. We used the number of clicks made by a participant during one trial as an estimate for the number of adjustments made, and in turn the effort put into the task. From this, we examined if the number of adjustments affected the final, accepted value.

When running a linear regression with a inverse proportional fit, we found a small but significant relationship between clicks and accepted delays (Static stimuli: *accepted delay* $= 40.5 + 300.7/x, R^2 = 0.03474, p < .001$ Dynamic stimuli: *accepted delay* $= 29.5 + 579.6/x, R^2 = 0.1058, p < .001$). In other words, higher numbers of adjustments were associated with lower accepted delay values. Figure 4.5 includes a scatterplot to illustrate this interaction, where the visible horizontal bands in the data can be attributed to the limited temporal resolution of the adjustment knob. The curve represent the regression line. We see that for few clicks, accepted delays range far up, while when the number of clicks increase we, accepted

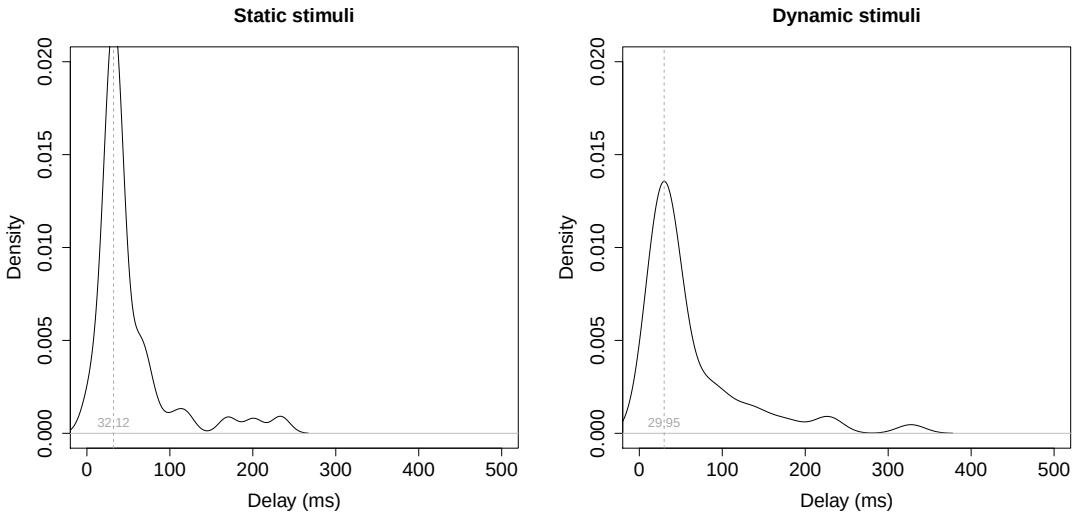


(a) Static stimuli



(b) Dynamic stimuli

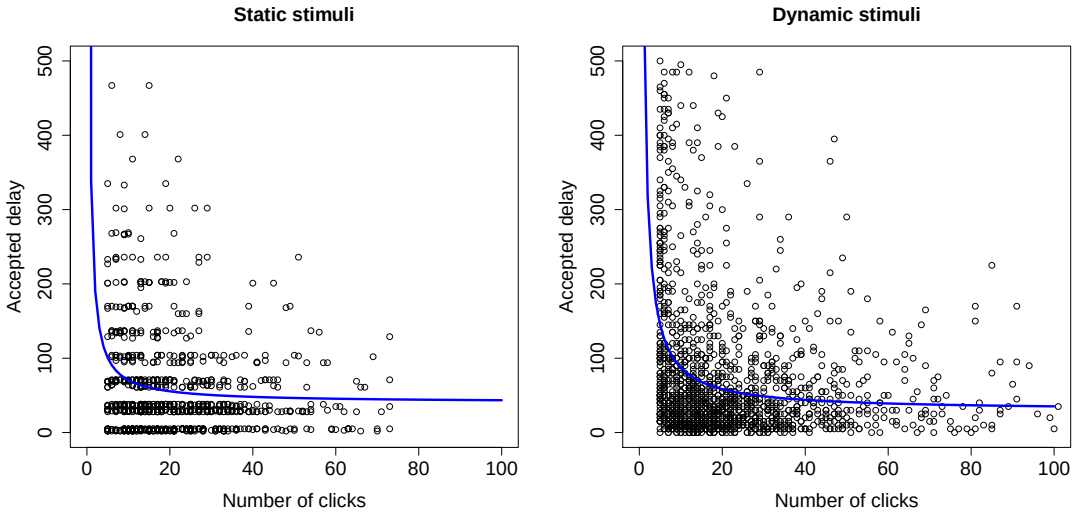
Figure 4.3: Individual adjusted delay scores plotted as an empirical cumulative density distribution. The x-axis shows participants' final accepted delay score, and the y-axis corresponds to the proportion of scores that fall within defined range.



(a) Static stimuli (b) Dynamic stimuli

Figure 4.4: Density plot of each participant's median accepted delay.

values converge to a lower value.



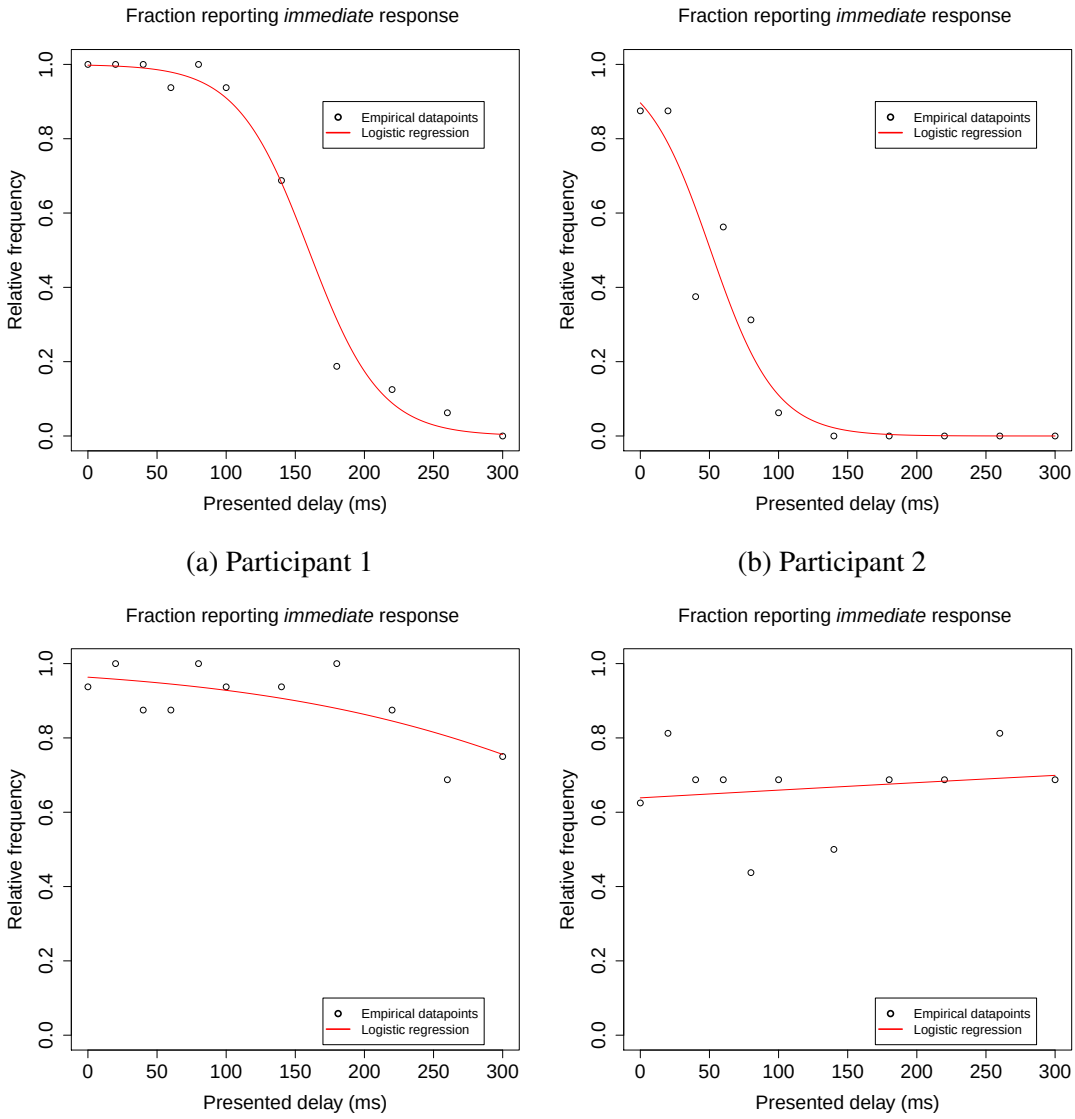
(a) Static stimuli (b) Dynamic stimuli

Figure 4.5: Density plot of each participant's median accepted delay.

4.4.2 Results from the discrimination task

The discrimination experiment uses what Gescheider (1976, pp. 46-50) describes as the *method of constant stimuli*. The data is a list of presented delays paired with responses of *simultaneous* or *delayed*. Ideally, a participant would answer only *simultaneous* for low delay values, some ambiguity in a small range until they clearly notice the delay and consistently answer *simultaneous*. The method of constant stimuli results in an s-shaped curve. We chose to use the sigmoid function, an s-shaped curve that can be fitted to the data using regression. Gescheider defines

the *threshold* as the delay where the participant would be equally likely to answer both options. While most participants' results fit with this model, there are some results that do not.



(a) Participant 1

(b) Participant 2

(c) Participant 3

(d) Participant 4

Figure 4.6: Individual participants' results from the discrimination experiment with logistic regression fit. We see clear differences in how participants handle the task. Participants 1 and 2 show a clear crossover from noticing to not noticing when delay increases. Participant 3 on the other hand seem unable to notice any of the delays we presented. Lastly, participant 4 seems to have answered more or less randomly.

To clarify this situation, we have plotted four individual participants' results in figure 4.6. Figure 4.6a shows a clear example of the expected situation, and the data fits very well with the logistic distribution ($r^2 = 0.92$). Next, figure 4.6b shows data from a participant who seems very sensitive. There is no initial plateau, which would probably be in the range of the system delay, but the rest of the curve fits nicely ($r^2 = 0.97$). On the other hand, we have participants such as the one whose data is plotted in figure 4.6c. It seems quite clear that this participant does not notice delays before we approach the highest presented values. We do not have data above

300 ms, but extrapolating the logistic regression gives us a threshold of 442 ms ($r^2 = .34$). We kept this and similar data in the set for further analysis; it is difficult to come up with a clear criterion for which individuals to exclude. Lastly, we have the result shown in figure 4.6d. The responses here seem more or less distributed around equal probability for both answers for all delays, and the regression line ($r^2 = 0.44$) actually has a positive slope, giving a *negative* value for the threshold. We assume this participant has answered randomly and exclude the data.

For the discrimination scores, we derived a best-fit logistic regression model, see figure 4.7 ($r^2 = 0.46$). Running another two Wilcoxon signed rank-sum tests, we found no significant differences between the rotation directions and the rotation speeds. Hence, we collapsed scores across presentation modes and established the discrimination threshold from the mid-point between *immediate* and *delayed* responses, at 148 ms. As before, we established the mode value for the discrimination mid-points from their density plot, which is illustrated in figure 4.8. Again the mode yielded a lower value than the median, this time it approximated 116 ms. The distribution showed a wide dispersion of scores around the mode, along with a long tail that ran to the beyond the experimental range of 300 ms.

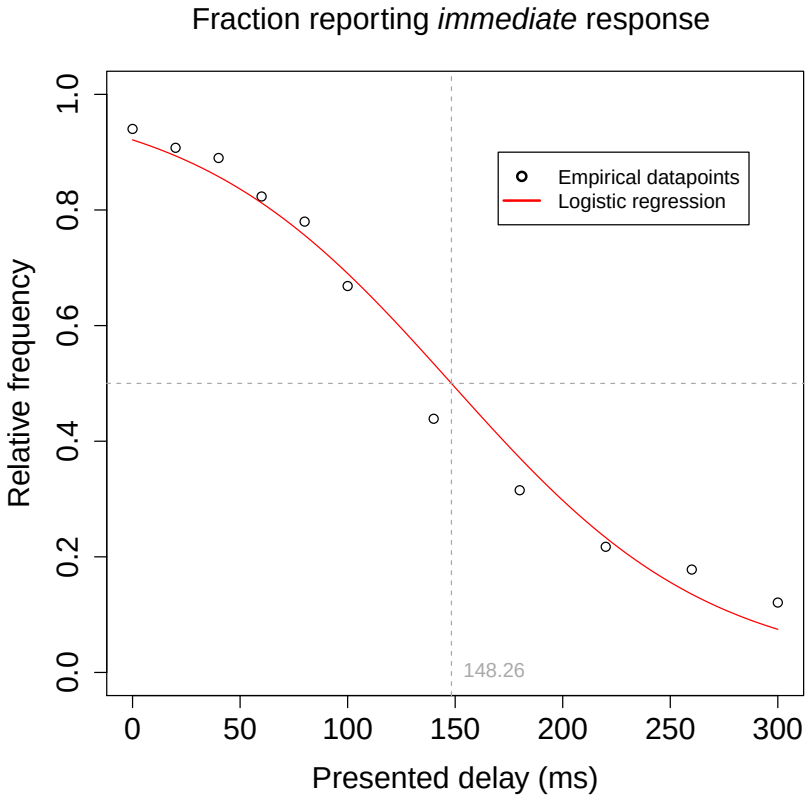


Figure 4.7: Results from discrimination task. The best-fitting logistic regression line with the proportion of participants' *immediate* responses plotted as a function of presented motor-visual delays.

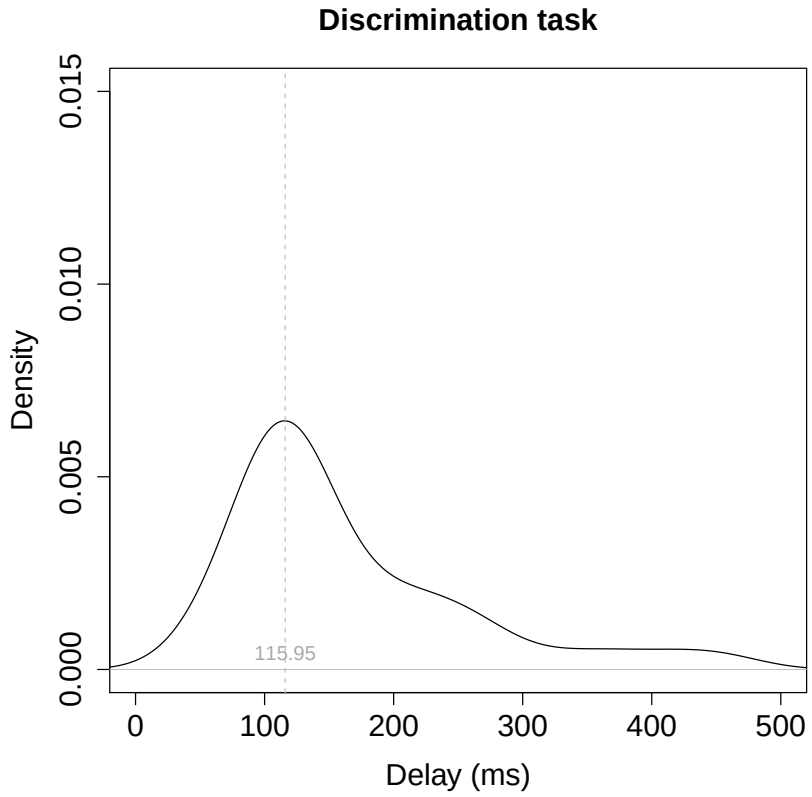


Figure 4.8: Density plot of each participant's sensory threshold in the discrimination task.

4.4.3 Comparison of experimental methodologies

This work includes two experimental methodologies to investigate the same issue: Human sensory threshold for delays between motoric output and visual input.

The adjustment task asks participants to adjust delay until they no longer notice it. The median result for this task is 40 ms and the mode is 30 ms.

The discrimination task presents participants with different delays and asks them if they noticed delay. The median result for this task is 148 ms and the mode is 121 ms.

Running a Wilcoxon signed rank-sum test, we settled that the striking difference between the mid-point thresholds established from the two methodologies is statistically significant ($W(49) = 1871, p < 0.001$). Furthermore, scores varied greatly across participants in both experiments. Yet, the wider dispersion of the density plot for discrimination mid-points suggests more individual variation for this task compared to the adjustment task. While the adjustment tasks allow participants to retry until they are satisfied, the discrimination task relies on participants paying close attention to individual presentations, which could explain the differences in variation.

For the moving stimulus, where the same participants participated in both experimental procedures, there is a clear correlation between individual participants' result in the two different experiments. Figure 4.9 shows a scatterplot of the two experiments and a linear regression line ($r^2 = 0.46$).

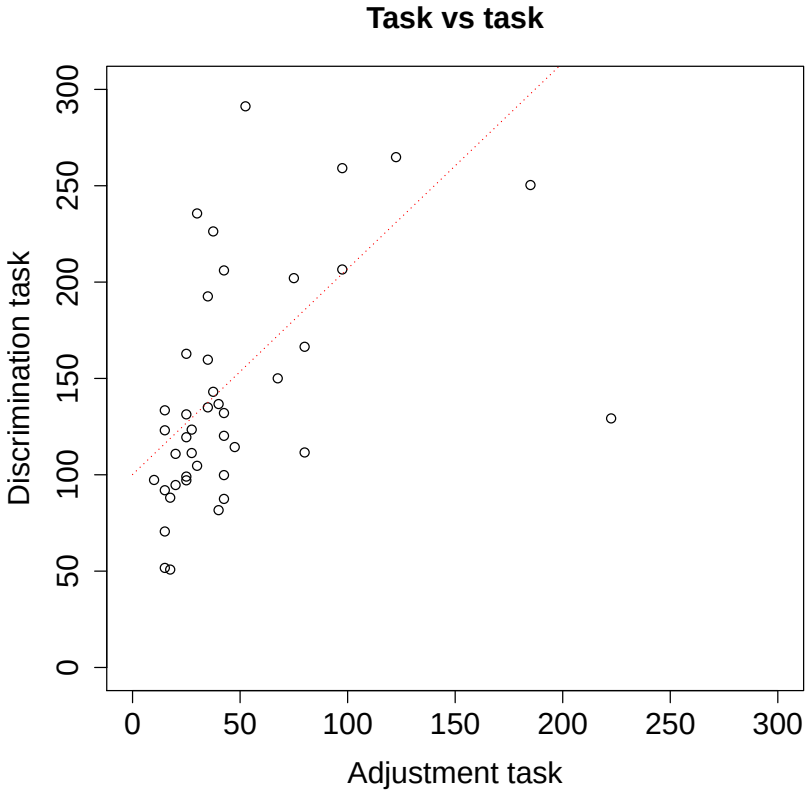


Figure 4.9: Discrimination task vs adjustment task for moving stimuli.

4.4.4 Gaming experience

From our motivation to explore individual differences, we collected data on participants' familiarity with gaming to explore possible correlations between their experience and temporal delay sensitivity. From these results, we divided participants into a high-experience and a low-experience group. Using the Wilcoxon test, we found no significant correlation between gaming experience and delay sensitivity in the adjustment task, for neither type of stimuli. (static stimuli: $W = 189, p > .3$, dynamic stimuli: $W = 183.5, p > .4$).

In the discrimination experiment on the other hand, results were indicative of a difference ($W = 127, p < 0.05$), with median threshold for the experienced group at 123 ms versus 206 ms for the inexperienced group. Investigating the individual game categories, we found that experience with *first person shooters*, *massive multiplayer online games* and *multiplayer online battle arenas* all indicate lower thresholds with p-values less than 0.1. The other game generes show no relation to delay detection thresholds.

4.4.5 Music experience

Another background detail we asked about was music experience. We asked the participants to name instruments and not how many years they had practiced music. Because we had relatively few participants very experienced in music, we divided them into groups with no experience and any experience. Using this division, we found significant differences for the moving stimulus experiments (discrimination task: $W = 405, p < 0.05$ location difference=48, adjustment task

task: $W = 372, p < 0.05$,location difference=20). The static stimulus experiments had a different set of participants, which did not show this correlation.

4.5 Discussion

The presented delay thresholds derive from our experiment on the interplay between human actions and screen responses. We highlight the applicability of these thresholds by relating them to relevant development of games.

4.5.1 Motor-visual temporal interactions

Several studies have already looked into the tolerance to motor-visual delays from an applied perspective, using full games, as we have seen in section 4.1. Others have studied the basic human ability to perceive delay. We are working towards unifying the two approaches. In this chapter, we set out to explore the detectability of these delays under the most ideal conditions, using simple, isolated stimuli. None of our experimental variables affected results significantly. These variables are initial delay, size of the disc, speed of rotation or direction of rotation.

All data presented in the section 4.4 represents delay deliberately added by the experiment software. This makes the numbers roughly comparable to numbers reported in earlier research on delay in computer games, such as that described in sections 4.1.1, 4.1.2 and 4.1.3. In psychophysics, researchers consistently report delays including inherent delay in the experimental setup or they build custom equipment that minimises delays. This means that to compare our numbers to numbers from 4.1.4 we need to add the delay introduced by our setup, as described in 4.3, measured to 51 ms.

For the adjustment task, the numbers represent the recorded delay duration at the moment of a participant's acceptance. So far, we have used median value as representative of these results, but is this the best quantifiable results of these responses? We can consider an *ideal* participant: One who consistently adjusts if a presented delay is above their threshold and consistently accepts if a presented delay is below their threshold. This ideal participant would not accept any delays above their detection threshold, but would accept any delay *below* this value. Hence, the *highest* value recorded should be the one closest to this hypothetical participant's threshold. Assuming each adjustment is of random size, accepted values would be distributed uniformly from 0 to the ideal participant's threshold, implying that the median value would be *half*. A real participant would not have such a stark cutoff point, but it is reasonable to assume that their median would be lower than their threshold. For the discrimination experiment, the numbers represent the presented delay, and no such effect can occur.

When incorporating these limitations into our motor-visual delay thresholds, we find that in the adjustment experiment, median accepted delay for the static stimulus to be 174 ms and for the dynamic stimulus it is 182 ms. In contrast, for the discrimination experiment we find a median threshold of 199 ms. If we based the results on mode values rather than medians the corresponding numbers would be 166 ms, 162 ms and 167 ms respectively. Taking these limitations into account, the striking difference between the two methodologies disappear, and a Wilcoxon test now reveals no difference ($W = 1000, p > 0.6$).

Although these thresholds allow room for uncertainty, they serve as guidelines to the sensitivity of the human perceptual system when encountering motor-visual delays. Considering the numbers reported above represent when a majority of our participants could detect delays a value well below this should be chosen as a design goal. The 25th percentile of both experiments is at 152 ms for detected delay, including system delay, and could possibly form a basis for developing guidelines. For the purposes of the rest of this thesis, we round this number to 150 ms. This number includes all sources of delay in a system: input devices, drivers, operating system, application, graphics card and display.

4.5.2 Perception of delay in games

As discussed in sections 2.1 and 4.1, delays can have serious impact on user experience in games. When optimising games, developers need a clear boundary on much delay they can accept. Traditionally, these boundaries have been defined by subjective tests initiated by the developer, or from rules-of-thumb based on experience. The current situation can certainly benefit from empirical findings on acceptable motor-visual delays. The presented work does not derive from a game, but from an extremely simplified user interaction. How does this interaction relate to games? At the most abstract level, games can be considered a series of user interactions. Much content in fast-paced games requires speedy reactions to the game world, but equally important are the immediate game responses that convey the results of an action. By establishing a lower bound for perceptible delay, game developers can instead rely on tested-and-tried thresholds. This introduces a level of confidence in the development stage, stating that *shorter* delays are unlikely to affect the conscious game experience. Although users will not be consciously aware of delays lower than 150 ms, actual gaming performance may still be negatively affected by imperceptible delays, an issue we have not investigated.

In cloud gaming scenarios, all network latency appears as interface latency. This means all actions are delayed by the network latency in addition to local delays. These can easily add up to more than the delays many of our participants detected, thus degrading the experience. This makes the presented experiment a good representation of current technological challenges. The most pronounced difference between real-life cloud games and our experimental scenario lies in the nature of the stimuli. In games, many tasks require reactions to moving objects and changing directions. Despite the contrast with our isolated disc presentation, we consider our results indicative of the shortest delays that would be noticeable in cloud gaming scenarios.

Client-server games behave quite differently. The difference arises mainly from the delay compensation techniques, which means that these games have two types of delay: Local interface delay and client-server delay. A player's actions do not need to be sent through the network, so the immediate effects of these come through with only local delay. In many cases the *consequences* of the actions are delayed by the full round-trip delay. This means that if you shoot, the muzzle-flash and recoil animation trigger after only local delay, while the effects on the target are delayed one full network roundtrip plus interface delay. Because of the additional complication in client-server games, our result do not apply directly to such games.

4.5.3 Perceptible delays compared to measured delays

In chapter 3, we measured how much delay there is between input and output in some current systems and found delays varying from 5 ms to 172 ms depending on hardware and software configuration. How do these delays compare to the observable delays?

Using the value 150 ms from section 4.5.1 as a reasonable estimate for detectable delays, we see that some combinations of system and settings produce delays that, according to our research, are long enough to be detected by players. Especially, the TV-screen can be considered too slow to use for many games. Note that this does not mean all TVs are too slow, some have *game modes*. Further, even if the local delays themselves are not long enough to be a problem, these delays add to any other delays, potentially pushing total delay above the threshold.

Interestingly, previous work does not discuss local delay in the way we do. None of the papers from the field of gaming cited in section 4.1 mention actual delay present in their setup, only delay added by their experimental conditions. This makes it very difficult to compare results between studies, because for a given experimental condition, we can not assume that total delay is equal. This would not be a problem if the measured local delays were insignificant compared to the delays introduced in the experimental setups. However, the values we have found for local delays indicate that these are at the same order of magnitude as the delays added in the experiments, sometimes even larger.

4.5.4 Latency compensation

Network latency can to a certain degree be compensated for, as described in section 2.4.2. Our experiments introduce uncompensated delay, and thus do not investigate how effective these techniques are. However, none of these techniques can compensate for local delay, which chapter 3 shows can be significant. Our results can be directly compared to measured delay in a local game to estimate if delay will be noticeable. For latency compensated client-server games, the situation is more complex. Delay in tese games consist of two components: uncompensated local delay and compensated network delay. Adding to the complexity is that commercial games use many distinct, often undocumented forms of latency compensation. Consequently, much work remains to determine the interaction between uncompensated delay, compensated delay and various forms of delay compensation.

4.6 Summary

In this chapter we have looked at how humans perceive delays in interactions with computers. Specifically we focused on how short delays were consciously noticeable in simplified interactions. Using two different methodologies we found a threshold of about 150 ms, but also large individual variations.

With these numbers in hand, a natural next step is to look to technical improvements to allow more complex games to achieve the target delay numbers.

Chapter 5

Game servers in the cloud

Having investigated how much delay players tolerate as well as how much delay is added locally we look at an important trend in game technology: Owning or renting servers on long-time contracts has more and more come to be considered a liability. Cloud computing, which allow renting of computing resources on an hourly, daily or monthly rate and very quick up- and downscaling is advertised as a solution to these problems. Microsoft[1] deliver their newest console with a cloud offering for developers. High-profile games such as *Titanfall*[2] and *Forza Motorsport 5*[3] already utilise this offer. This chapter attempts to answer question three from section 1.2: "Are cloud services responsive enough to run game servers?" and is based on work presented in "Is todays public cloud suited to deploy hardcore real-time services?" (Paper VI, p.155).

5.1 Related work

Service providers rarely control their own infrastructure, and finding the right offer among the various providers is difficult. Traditional offers usually include long contract terms, limiting flexibility. This creates a dilemma where on one side, under-provisioning is safer in case the product does not sell as well as expected but limits the potential income from a successful release. Over-provisioning, on the other hand, allows full utilisation of a success but leaves providers with a huge amount of expensive infrastructure in case of lower sales. Cloud computing is a relatively recent concept for alleviating these issues. Clouds, however, give clients significantly less control over the running environment. Machines are mostly virtual, as opposed to earlier offerings where clients could get full access to physical machines. This makes clouds a challenging new environment for running interactive, response-time sensitive applications. Work on performance in clouds has so far focused on the typical large, bulk jobs of scientific computing or web servers. Scientific computing requires high performance averaged over timespans in minutes and hours. Web-servers need to serve as many request as possible during a given time, but each individual request has a relatively loose time constraint. Conversely, games require consistent response time in milliseconds. Even quite rare events with

[1] `http://www.wired.com/2013/05/xbox-one`
[2] `http://www.engadget.com/2014/03/10/titanfall-cloud-explained/`
[3] `http://www.engadget.com/2013/05/21/forza-5-coming-to-xbox-one-at-launch/`

slow response time might significantly affect perceived quality. Web servers need consistent response time, but the deadlines are considerably looser than for games.

Schad et al. (2010) analysed the Amazon EC2 cloud using a standard suite of standard benchmarks examining all aspects of the platform. Each time a benchmark was run, it started on a fresh virtual machine instance, allowing sampling of the instance pool. This benchmark ran every half-hour for a month. These showed a considerable variation in performance within the same instance type and location, as well as variations by day of the week. Ostermann et al. (2010) tested a somewhat larger suits of instance types, using typical scientific workloads. Their conclusions were much the same: Reliability was not sufficient for any configuration. El-Khamra et al. (2010) find significantly less variation using single core instances, but the variation grows with the number of cores. This discrepancy might be explained with the different types of workloads.

Closer to the topic of this chapter, Barker and Shenoy (2010) experimented on Amazon EC2 with what they termed *microbenchmarks*, benchmarks taking a few hundred milliseconds to run, and examined them for variations in runtime. First, they ran a CPU only benchmark. This showed a base runtime for the benchmark, representing the time it took to complete with full CPU access. Other samples were however much slower. Their results showed: "While most of the tests completed in roughly 500 ms, there was frequent variations between 400 and 600 and many outliers taking significantly longer; a few even took more than an entire second." These variations in runtime represent CPU availability, *jitter*, at a scale more than enough to disturb any game. Hard drive access times were even more variable and also showed the clear division in performance between undisturbed samples and shared resource samples. For networking, the results were less clear. Further, the authors compared these results with results from a laboratory setup using the Xen hypervisor. Lastly, they ran an actual game server, *Doom 3*, on the cloud computer instance, which resulted in very unpredictable performance.

What do we know about the usefulness of cloud services for interactive applications? Barker and Shenoy (2010) shows performance stability issues in Amazon EC2, though the details could be explored further. We can find no data from other cloud providers to draw any clear conclusions. Thus, we decided to investigate further.

5.2 Experiment Design

Barker and Shenoy (2010) proposed an experiment to evaluate how suitable cloud services are for time-sensitive applications such as games. However, they only investigated a single offering from one provider, and did not report detailed statistics. We decided to expand on this work, adding more offerings from more providers. Further, we report more detailed statistics. Our work isolates CPU performance, leaving network and other subsystems for further work. We compare response time stability, with respect to processor availability, between cloud services and natively running software, as well as between two different cloud providers. The result is a recommendation for services providers.

Our question does not focus on the absolute performance of different cloud service levels. Generally, costumers can buy the performance they need for their applications. However, stability of performance is not specified in the contracts and costumers who depended on stable

performance, such as game providers, need to get this information from other sources. Hence, the absolute average values of the benchmark runtime are not important, and we are only interested in the stability of the result. Neither does the load need to behave like any specific realistic load. This evaluation is only concerned with the actual availability of computing resources on a fine-grained time resolution.

With this in mind, the benchmark was designed as a small loop using array lookups and arithmetic, where the array is sized to be significantly larger than the cache size, this allows us to load both the CPU and the memory system. The purpose of this benchmark is not to evaluate the performance of the systems, but rather *detect* if the hypervisor stalls our virtual machine at any point in time. We examined the memory subsystem and the CPU performance in isolation, ignoring external I/O to limit the scope of the work. The tight loop of the benchmark is designed as a *worst case* load scenario for a virtual machine. Each machine was tested with a number of threads equal to the number of cores available to the virtual machine. Note that each instance of the benchmark is single-threaded and does not show better results by increasing the number of cores. For these reasons, the most relevant single metric is the *coefficient of variation*. This value is defined as $c_v = \frac{\sigma}{\mu}$, where σ is the standard deviation and μ is the mean.

In addition to the benchmark code, the utility *mpstat* was run in the background. Mpstat gives us *% steal*, a metric that reports how much of the physical CPU is unavailable to the current OS instance because the hypervisor has assigned it to other instances.

Although our concern is to investigate the variance in execution time for the benchmark, a by-product is that we get an indication of the processing power provided by the different providers/service levels. This may also be of interest to game providers considering cloud deployment.

The benchmark was run on three different instances in Amazon EC2, in the zone *us-east-1d* (Table 5.1), as well as three different instances in Windows Azure (Table 5.2).

Micro Instance	613 MiB memory Up to 2 EC2 Compute Units (for short periodic bursts) I/O Performance: Low	USD 0.020 per Hour
Medium Instance	3.75 GiB memory 2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit) I/O Performance: Moderate	USD 0.120 per Hour
High-CPU Medium Instance	1.7 GiB of memory 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each) I/O Performance: Moderate	USD 0.145 per Hour

Table 5.1: Technical specifications from Amazon (Amazon, 2013).

Extra small instance	768 MiB memory 1 shared core	USD 0.020 per Hour
Small Instance	1.75 GiB memory 1 CPU Core	USD 0.085 per Hour
Medium Instance	3.5 GiB memory 2 CPU Cores	USD 0.160 per Hour

Table 5.2: Technical specifications from Microsoft (Microsoft, 2013).

To compare the different instances, the benchmarks were run for 18 hours each under the configurations shown in tables 5.1 and 5.2. To create a baseline for comparing the cloud services with native dedicated hardware, the same benchmark was run on a reference system, a

standalone PC [4] without any virtualisation, chosen to be approximately the same class of performance as the cloud computers.

Schad et al. (2010) suggest that there might be differences in performance of the cloud systems based on when the benchmark is run, either by time of day or day of week. To investigate this, we ran the benchmark described above for three minutes every half hour for a week (the week of 27th of May 2013). Between each run, we stopped the instance and started it again before the next run. According to Schad et al., this should allow us to get a different physical machine every time. If instances of the same category have different properties, this allows us get a somewhat random distribution of instances.

The Linux kernel reports a metric called *CPU steal*. This represent the proportion of CPU cycles that the hypervisor reserves for other instances running on the same CPU. A large CPU steal number indicates that less processing is available for the current operating system, and impacts performance. We recorded this number for all experiments.

5.3 Evaluation

To show the baseline of consistency we can expect from this benchmark while running natively, we included a reference system. Deviations from this baseline can be assumed to result from the hypervisor or other aspects of the cloud system. In this system, the benchmark reports very consistent results. The average runtime is 71.8 ms with a standard deviation of 1.52, giving a coefficient of variation (c_v) of 0.02.

5.3.1 Amazon EC2

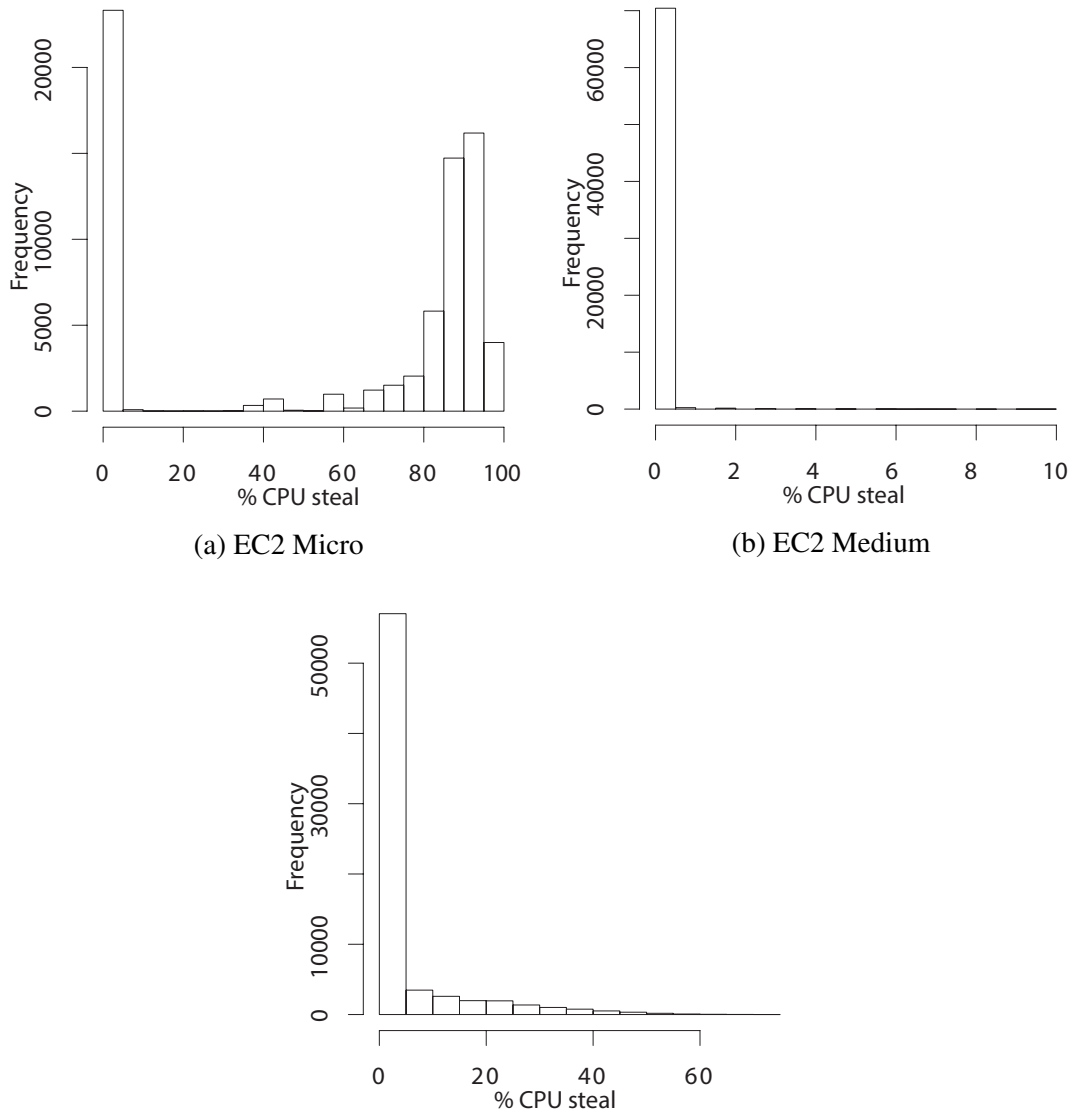
The *CPU steal* metric is only sampled once a second, and so has a low resolution, but with sufficient samples, we build up a picture of how much CPU is allocated to each instance, as well as the distribution over time.

The first thing to note in this data (figure 5.1) is the fact that all the instance types have a significant number of samples where the current instance has full use of the CPU. These situations pull the average CPU availability significantly up, allowing the provider to fulfil their obligations despite other samples with much less available resources.

The next point standing out is the *medium* instance. According to the pricing information, the available CPU resources on this instance type equal to exactly one core of the current hardware. This seems to create a situation where our instance gets full use of the core most of the time, with very little steal. Increasing the available CPU power to the *high cpu* instance adds another core, making two cores available for the virtual machine to use. However, the instance only has access to part of this core and the CPU steal value increases again.

To visualise the results, we plotted benchmark run-times as histograms in figure 5.2. These show several interesting values. First of all, for both the *micro* as well as the *medium, high-CPU* instance, there are two clear peaks. This matches the CPU-steal data, where the low values represent the situations where our instance got full use of the CPU, and the high run-times are the result of periods of higher CPU-steal. Again we see the advantage of the *medium* instance,

[4] Intel Core 2 Duo CPU E7500 @ 2.93GHz, 4GB RAM



(a) EC2 Micro

(b) EC2 Medium

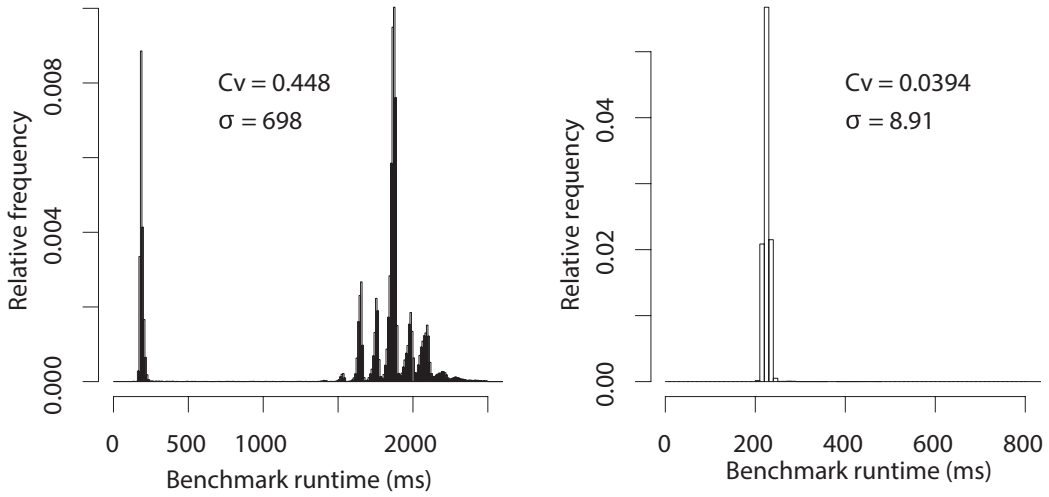
(c) EC2 Medium High CPU

Figure 5.1: CPU Steal histograms. Samples every second.

in that it has a much more stable runtime, with c_v equal to 0.04. This more stable instance type depends on the fact that it allocates exactly one core to the instance. As the processing power granted each instance type is defined by an abstract, hardware independent metric, clients have no guarantee that this situation will continue indefinitely. Rather, when the underlying hardware is upgraded, it is very likely that each core will provide more power, and the currently stable instance type will become unstable.

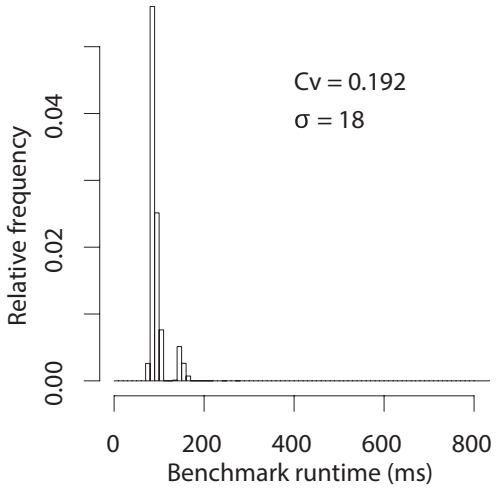
5.3.2 Microsoft Azure

On the *Microsoft Azure Virtual Machine* cloud, the operating system always reports zero CPU steal, as if it was running on its own dedicated machine. This implies either that the hypervisor hides these details from the operating system, or that our instance actually has a dedicated CPU resource.



(a) EC2 Micro

(b) EC2 Medium



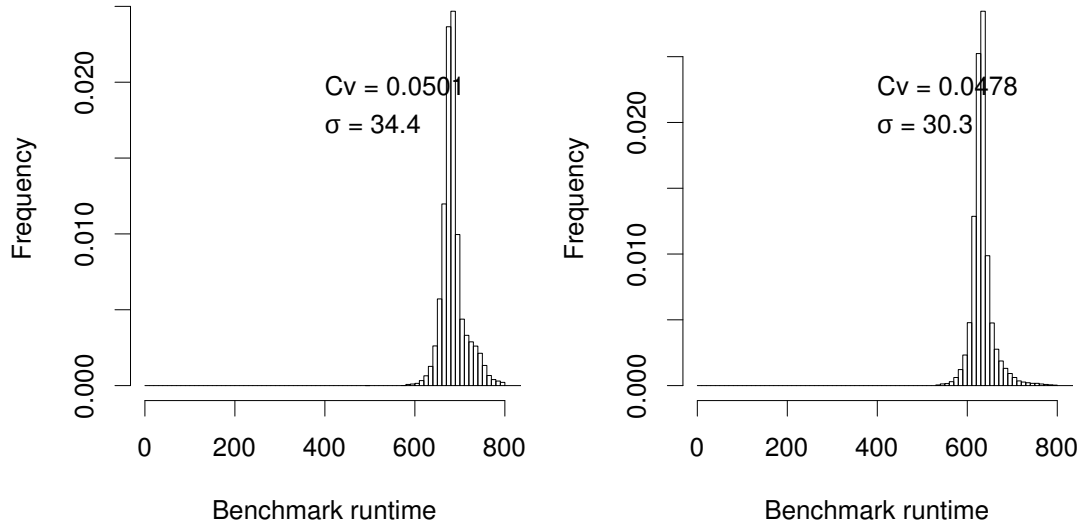
(c) EC2 Medium High CPU

Figure 5.2: Amazon EC2: Histogram of benchmark runtimes. Note different scale on the axis of 5.2a.

Compared to the Amazon case, access to the CPU is significantly more stable in the Azure cloud (figure 5.3). Regardless of instance type, the runtime of the benchmark is almost completely predictable and stable, c_v is 0.05 for all instance types. The deviation is however twice that of the reference case. Depending on the exact level of time-sensitivity of the load, and its computational cost, this could be acceptable or not. The single-threaded nature of the benchmark explains that the *small* and *medium* show almost identical results.

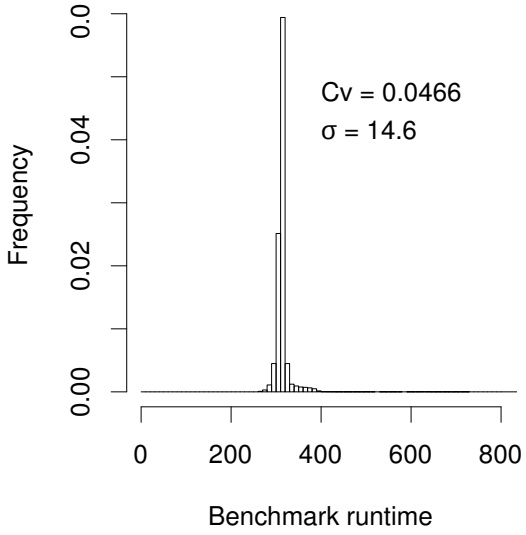
5.3.3 Time series

Figure 5.4 shows the results of running the benchmark repeatedly over a week on an *EC2 Medium High CPU* instance. Three bands of performance are visible throughout the week, and there is no clear difference based on weekday or time of day. About the reasons for these bands



(a) Azure Extra Small

(b) Azure Small



(c) Azure Medium

Figure 5.3: Microsoft Azure: Histograms of benchmark runtimes.

we can only speculate. They possibly represent our load being swapped out an integer number of times before completing.

5.3.4 Overview

Putting it all together, figure 5.5 shows some differences between the providers. The low-end instance from Amazon has variations far above the acceptable level for any time-sensitive application. If a server needs 20 ms on average for processing a player client, this time would occasionally increase to 10 times that value, exceeding acceptable latency for most games as described in chapter 4. Interestingly the *Amazon EC2 Medium High CPU* is also quite unpredictable, though this configuration is sold as higher performing than the *Medium* type. Among the Amazon offerings examined only the *Medium* instance type is near acceptable. From Microsoft Azure, all instance types are reasonably stable in performance. All show variations

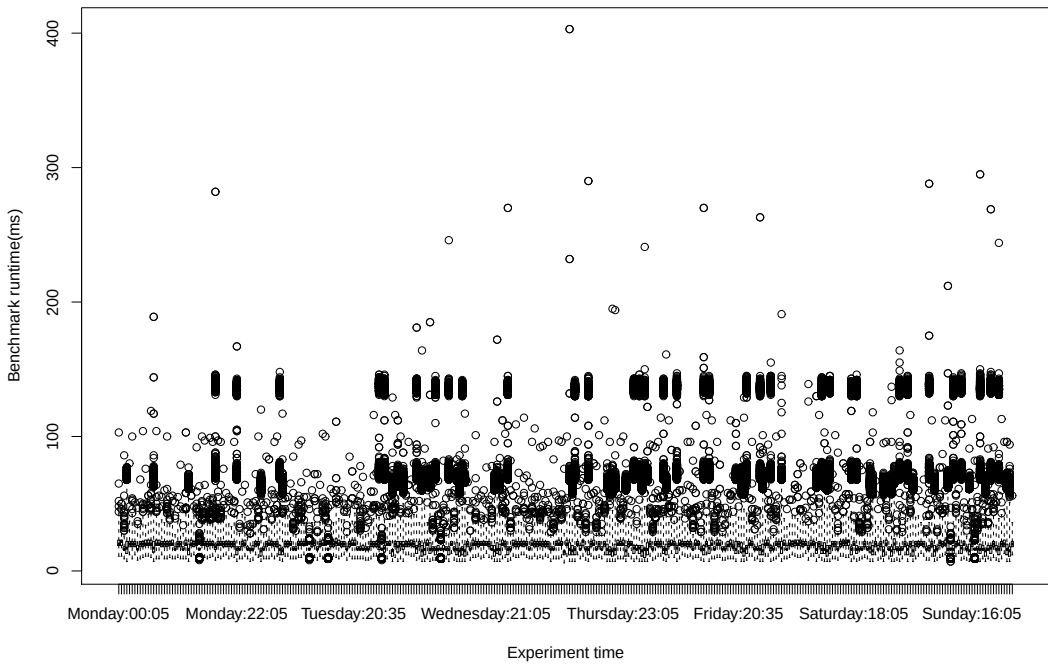


Figure 5.4: Time series of Amazon EC2 Medium High CPU.

above our native machine reference. The significance of these variations depends on the requirements of the application. They are small enough that they should only interfere with very demanding games.

5.4 Summary

A current trend is to put all services in the cloud, and games are no exception. In this context, we found that cloud services are convenient and in many cases cost-effective for running game servers. However, they are not yet stable enough for fast paced games. We see random glitches in performance where the servers seem to hang for hundreds of milliseconds. As we found in chapter 4, delays of more than 150 ms are detectable by players. Note that we have only tested two cloud providers, and the numbers are somewhat outdated. Cloud providers have increased their priority on games as a class of application, and new classes of service specifically focused on games are on their way.

Although cloud computing is sold as a new concept, at the core, it still uses traditional server parks. Large numbers of servers connected to a network run an even larger number of virtual machine instances. Each of these instances is not usually very powerful. For really heavy loads, this might not be ideal. A large game world hosting hundreds if not thousands of players could benefit from a more powerful integrated server architecture.

Next, we will look at how to implement such a large-scale software architecture.

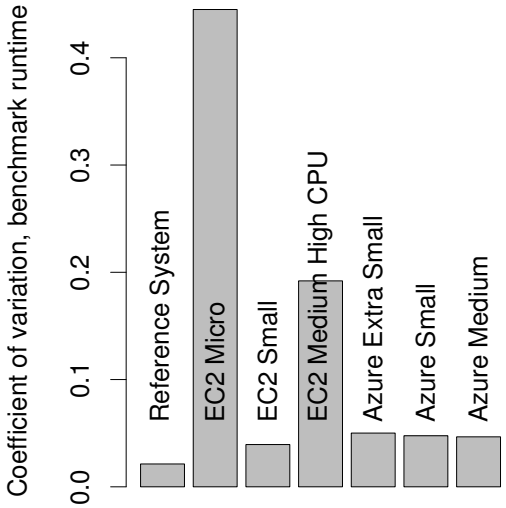


Figure 5.5: Summary of benchmarks for the different systems.

Chapter 6

Parallelising game servers

In chapter 3, we measured how much delay there is in computer games, and in chapter 4, we investigated how much delay players may tolerate. Most of the delay measured arises in hardware and drivers, outside the control of developers, making this a difficult target for improvement. Others have looked at optimising networks for games (Petlund, 2009; Hayes et al., 2015). To expand on the previous knowledge, we focus on the server side of the game pipeline.

Many server loads, such as web servers, are inherently parallel: Each user can be served independently of the others. Game servers, on the other hand, are designed to allow realtime interaction between clients. To implement this important feature, a game server needs updated information about many clients simultaneously. Thus, the standard paradigm of assigning an independent thread on the server to each client can not be used for game servers without extensive synchronisation and locking. This chapter suggests one possible answer to question 4 asked in section 1.2: "How can parallelism be better utilised in game servers for massive multiplayer games?" and is based on work presented in "LEARS: A Lockless, relaxed-atomicity state model for parallel execution of a game server partition" (Paper VII, p.167).

6.1 Motivation

Current Massively Multi-player On-line Games, described in section 2.2.2, allow about a hundred people to congregate in one area. *World of Warcraft* for example allows two teams of 80 players to compete in a single battle[1]. Simulation of interactions between these players requires large amounts of processing. In the future, the number of concurrent players might be much greater, and the simulation will be more detailed and sophisticated. To make these developments preform acceptably to the players, the simulation has to be updated frequently and at a consistent rate. This means that developers have to deal with deadlines as well as processing requirements. As described in section 1.1.4, to utilise modern hardware in scaling games, developers have to take advantage of the highly parallel nature of modern systems.

Currently many servers utilise a single loop to handle all clients in one area. The loop reads updates from all players, processes their actions and sends the results to each participant. Different areas can be processed by different threads, but clients cannot interact with those processed in another thread.

[1] `http://us.battle.net/wow/en/game/pvp/battlegrounds/tol-barad`

One major goal for large game providers is to support as many concurrent players in a game-world as possible while preserving the strict latency requirements in order for the players to have an acceptable quality of experience (QoE). To achieve this, game-worlds are typically partitioned into areas-of-interest to minimise message passing between players with no interaction and to allow the game-world to be divided between servers. This approach is however limited by the distribution of players in the game-world.

To extend such servers beyond what a single thread can handle, many developers rely on *sharding*. *Sharding* involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. Chu (2008) investigated the possibilities and limitations of this approach, which is widely used in current games.

By the classification in section 2.3, partitioning games into areas-of-interest or shards only allows scaling of the *overall number of simultaneously active users*. Our goal is to improve the third, and most important class of scalability, *player density*, or the number of people who can directly interact simultaneously.

Different benchmarks can be employed to measure how successful an implementation of a game server architecture is. We focus on how the system copes with increased player density by measuring latency. The accumulated latency of network transmission, server processing and client processing adds up to the latencies that the user is experiencing, and reducing any of these latencies improves user experience.

6.2 Related work

For many traditional online games, distribution of work is simple. If there are too many players for the hardware to handle, the provider or players themselves can set up a new server, allowing new players to connect there. Players on the different servers are in this scenario unable to interact. Workarounds exist, such as transparent shifting of players between servers or the ability for simple cross-server communication such as chat, but fundamentally, these are the same design. For many modern games, this approach is not acceptable. Game designers and players want more users interacting in increasingly complex ways.

6.2.1 Binary Space Partitioning between threads

Abdelkhalek and Bilas (2004a) suggested a significant improvement to the traditional approach. They assigned each player to a separate thread on the server, optimally allowing parallelism to scale with the number of players. To test large numbers of simultaneous players they used automated players, bots. For interactions between players, or players and the world, this approach requires synchronisation. Because the server splits the world in smaller partitions for other reasons, the system locks only the partitions affected by the action. These partitions are created using the *Binary Space Partitioning* (BSP) algorithm, which divides each section of the world in two pieces every time the section gets too crowded. This synchronisation is also the major bottleneck of the system. From the conclusion:

> However, the game processing component of the request processing phase incurs

> high synchronisation overheads, mainly due to lock contention for shared game objects. Locking overhead is up to 35% of total execution time.Abdelkhalek and Bilas (2004a, p. 10)

6.2.2 Transactional approaches

Building on a binary space partitioning, Gajinov et al. (2009) implemented a version of the Quake server based on *transactional memory*, which allows code do be marked as transactions and run speculatively. Both these systems were experimental at the time of writing. If there is a conflict, the transaction was automatically rolled back. This paradigm requires hardware support, as well as support from compilers. Using a heavily modified game server as a suggestion for using transactional memory, the authors showed that some concepts were considerably easier to express with transaction semantics than traditional locks. Actual performance, however, was hampered by frequent rollbacks, as well as the experimental nature of the underlying technology.

The *Red Dwarf* project, the community-based successor to *Project Darkstar* by Sun Microsystems and investigated by Waldo (2008), is another good example of a parallel approach to game server design. Here, response time is considered one of the most important metrics for game server performance, and uses a parallel approach for scaling. The described system uses transactions for all updates to world state, including player position.

6.2.3 Improved Partitioning

Beskow et al. (2009) investigated partitioning and migration of game servers. Their approach used core selection algorithms to locate the most optimal server.

Abdelkhalek and Bilas (2004a) discussed the behaviour and performance of multi-player game servers. They found that in terms of benchmarking methodology, game servers were very different from other scientific workloads. Most of the sequentially implemented game servers could only support a limited numbers of players, and the bottlenecks in the servers were both game-related and network-related. Abdelkhalek and Bilas (2004b) extended their earlier work and use the computer game Quake to study the behaviour of the game. When running on a server with up to eight processing cores the game suffers because of lock synchronisation during request processing. High wait times due to workload imbalances at global synchronisation points were also a challenge.

As we see here, some research exist on how to partition the server and to scale the number of players by offloading to several servers. Modern game servers have also been parallelised to scale with more processors. However, a large amount of processing time is still wasted on lock synchronisation, or the scaling is limited by partitioning requirements.

6.2.4 Databases as game servers

Researchers from the field of databases are also studying parallelising and optimising computer games. This perspective is very different from the ones mentioned previously. White et al.

(2007) propose implementing artificial intelligence scripts using a relational language. This description can then be processed using techniques from relational database optimisation.

This group also introduces the concept *state-effect pattern* in White et al. (2008). They test this and other parallel concepts using a simulated actor interaction model, in contrast to the practical approach of evaluating a running prototype of a working game under realistic conditions.

Treating games as a data management problem has some limitations though. Scripting is limited to a special purpose programming language that does, for instance, not support loops and has a both syntax and semantics unfamiliar to most game scripters. Further, the proposed approach is evaluated and recommended in situations where there are only a few different patterns of behaviour, but many units following each. It is not tested and seems tricky for situations with highly individualised behaviours.

6.3 Concept for parallelising game servers

As described in section 6.2, previous work has focused on splitting the load between multiple servers. All these approaches have the disadvantage of scaling mainly if players can be isolated in some way. Many production games simply isolate players completely from players in other areas. If too many players gather in one spot, either performance degrades, or the system splits the game world in separate *instances*. These are independent copies of the area, where nothing happening in one instance can affect the others. Both industry and researchers have analysed this scenario thoroughly. In contrast, we want to improve *player density* as defined in section 2.3. Thus, our work focuses on situations where every player can interact with every other player at any time. Restricting the case in these ways makes for a *worst case* scenario for game servers, a situation that has not been thoroughly studied.

6.3.1 Traditional approach

Traditionally, game servers are based around a main loop, which updates every *dynamic entity* in the game. These are entities in the game world that can perform actions independently. Dynamic entities include projectiles, player characters and non-player AI controlled opponents (hereafter referred to as *NPCs* for *Non-Player Character*). The simulated world has a list of all the dynamic entities in the game and typically calls an *update* method on each element, allowing it to perform its tasks. Simulated time is kept constant throughout each iteration of the loop so that all elements get updates at the same points in simulated time. This point in time is referred to as a *tick*. Since only one element updates at a time, no actual race conditions are possible. Thus, the traditional game server is a classical discrete-event simulation as defined in Jain (1991).

However, this does not mean that all issues of ordering are solved. Consider a shooting game. Player A and player B fire at a target simultaneously. Only one player can get points for this target. Who gets the point? Network latency is in most cases the dominating factor determining this. If the latency of one player delays their message so it arrives one tick later, the player with the lower latency gets the point. In many implementations, another influencing

factor is the order in which the players are processed. Assuming both packets arrive simultaneously, the player processed first wins. This is accepted and tolerated, and we consider this system to be the baseline from which to construct a new approach.

Further, single-threaded approaches make no attempt at keeping state consistent throughout an update. In the single-threaded main-loop approach, every update is allowed to change any part of the simulation state at any time. In such a scenario, the state at a given time is a combination of values from two different points in time, current and previous. Like the arbitrary ordering, this is accepted, and we consider this part of the baseline from which to proceed.

6.3.2 Relaxed constraints

Our concept is an extension of a concept proposed by White et al. (2008). They describe a model they call a *state-effect pattern*. Based on the observation that changes in a large, actor-based simulation are happening *simultaneously*, they separate read and write operations. Read operations work on a consistent previous state, and all write operations are batched and executed to produce the state for the next tick. This means that the ordering of events scheduled to execute at a tick does not need to be considered or enforced.

Our concept is based on this idea. However, because many aspects of game servers do not need to be deterministic, we remove three requirements on ordering and atomicity.

- We remove the requirement for batching of write operations, allowing these to happen anytime during the tick. The rationale for this relaxation is found in the way traditional game servers work. As mentioned in section 6.3.1, existing game servers make no effort to enforce separation between current and previous state.

- We note that execution order is already arbitrary, determined by factors outside the control of the game provider, so we see no need to enforce strict ordering internally.

- Further, we relax the requirements that game state updates must be atomic.

The fine granularity of games creates a need for significant communication between threads to avoid problematic lock contentions. However, game servers are not accurate simulators, and again, depending on the game design, some transient errors are acceptable without violating game state consistency. Consider the following example: Character A moves while character B attacks. If only the X coordinate of character A is updated at the point in time when the attack is executed, the attack sees character A at a position with the new X coordinate and the old Y coordinate. This position is within the accuracy of the simulation, which in any case is no better than the distance an entity can move within one tick.

We allow any dynamic entity to *read* the state of any other without locking. To *update* the state of other dynamic elements, they have to send messages. Such messages can only contain relative information and not absolute values. For example, if a projectile needs to tell a player character that it took damage, it should only inform the player character about the amount of damage, not the new health total. All changes to one dynamic entity are put in the same queue, and the one thread processes the entire queue. Thus, all updates are based on up-to-date data. This form of communication is extremely fast. Our current design is restricted to symmetric

multiprocessor machines, and all threads share the same memory. Thus, all communication between dynamic entities go through memory.

These relaxations allow actions to be performed on entities in any order without global locking while still retaining consistency for most game actions. This includes actions such as moving, shooting, spells and so forth. Consider player A shooting at player B: A subtracts her ammunition state, and send bullets in B's general direction by spawning bullet entities. The bullet entities run as independent work units, and if one of them hits player B, it sends a message to player B. When reading this message, player B subtracts his health and sends a message to player A if it reaches zero. Player A then updates her statistics when she receives player B's message. This series of events can be time critical at certain points. The most important point is deciding if the bullet hits player B. If player B is moving, the order of updates can be critical in deciding if the bullet hits or misses. In the case where the bullet moves first, the player does not get a chance to move out of the way. This inconsistency is, however, not a product of our approach. Game servers, in general, insert dynamic elements into their loops in an arbitrary fashion, and there is no rule to state which order is *correct*.

6.3.3 Limitations of our approach

While this system usually does not need transactions, they can be included, should a game include some actions where atomicity and ordering are critical. Examples of this include trading between players. Fortunately, as we have seen, most common game actions do not require transactions. The concept is also limited to symmetric multiprocessing systems. We make the assumption that any entity is able to read the state of any other entity without overhead.

The end result of our proposed design philosophy is that there is no synchronisation in the server under normal running conditions. Since there are cases where transactions are required, they can be implemented outside the main event handler and run as transactions requiring locking. We call this concept *a lockless, relaxed atomicity state model for parallel execution in game servers* abbreviated *LEARS*.

6.4 LEARS Design and Implementation

As a proof-of-concept of the described design philosophy we implement and test a simple game-server. The main concept is to split the game server executable into lightweight threads at the finest possible granularity. Each update of every player character, AI opponent and projectile runs as an independent work unit.

In our experimental prototype implementation of the LEARS concept, the parallel approach is realised using thread-pools and blocking queues. From traditional game server architectures, we retain the concept of *dynamic elements* as described in section 6.3.1, but now allow any dynamic elements to run concurrently.

6.4.1 Thread-pool

Creation and deletion of threads some incur overheads and context switching is an expensive operation. These overheads constrain how a system can be designed. Threads should be kept as

long as possible, and the number of threads should not grow unbounded. We use a *thread-pool* pattern to work around these constraints, and a thread-pool executor to maintain the pool of threads and a queue of tasks. When a thread is available, the executor picks a task from the queue and executes it. The thread pool system itself is not pre-emptive, so the thread runs each task until it is done. This means that in contrast to normal threading, each task should be as small as possible, and larger units of work should be split up into several sub-tasks.

The thread-pool is a good way to balance the number of threads when the work is split into extremely small units. When a dynamic entity is created in the virtual world, the thread-pool executor schedules it for execution, and the dynamic entity updates its state exactly as in the single threaded case. Furthermore, our thread-pool supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a time specified in the future. When the task is finished for one time-slot, it can reschedule itself for the next slot, delayed by a specified time. This allows dynamic entities to have any lifetime from one-shot executions to the duration of the program. It also allows different entities to be updated at different rates depending on the requirements of the game developer.

The same thread-pool executes all work, including the slower I/O operations. This is a consistent and clear approach, but it does mean that game updates could be stuck waiting for I/O if there are not enough threads available.

6.4.2 Blocking queues

The thread-pool executor used as described above does not constrain which tasks are executed in parallel. All systems elements must therefore allow any of the other elements to execute concurrently.

To enable a fast communication between threads with shared memory, we allow each thread to read freely from any data. For updates, we use *blocking queues*, which are message queues synchronised separately at each end. This means that messages can be removed from and added to the queue simultaneously, and since each of these operations is extremely fast, the probability of blocking is low. In our design, all dynamic entities can potentially communicate with all others. Thus, these queues allow information to be passed between dynamic elements. Each dynamic entity has a blocking queue of messages. During its update, it reads and processes the pending messages from its queue. Messages are processed in the order they were put in the queue and each queue is processed in one thread. This ensures that updates are always based on updated data.

6.4.3 Our implementation

To demonstrate LEARS, we have implemented a prototype game containing all the basic elements of a full MMOG with the exception of persistent state. The basic architecture of the game server is described in figure 6.1. The thread pool size can be configured, and executes the different workloads on the CPU cores. The workloads include processing of network messages, moving computer controlled elements (in this prototype only projectiles) checking for collisions between players and hits by projectiles before sending outgoing network messages.

Persistent state does introduce some complications, but as database transactions are often

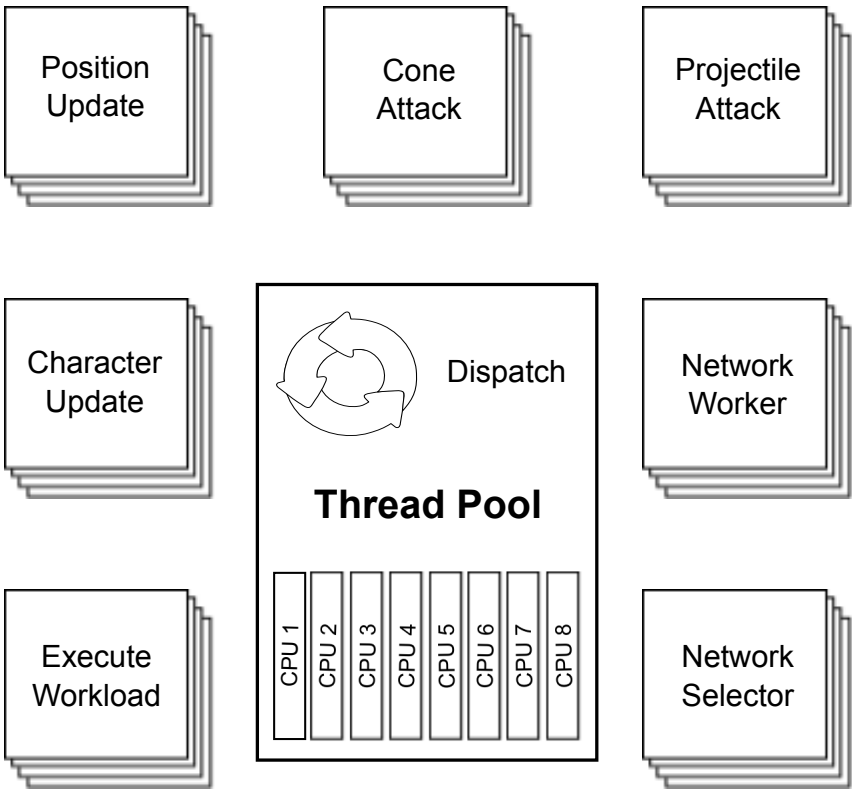


Figure 6.1: Design of the Game Server

not time critical and can usually be scheduled outside peak load situations, we leave this to future work.

The game is an extended, multiplayer version of *Pacman*. In this game, each player controls a small circle representing *the character* with an indicator for which direction they are heading as shown in figure 6.2. Pressing keyboard buttons moves the characters around. They also have two types of attack, one projectile and one instant area of effect attack. Both attacks are aimed straight ahead. If an attack hits another player character, the attacker gets a positive point, and the character that was hit gets a negative point. The game provides examples of all the elements of the design described above:

- The player character is a long-lifetime dynamic entity. It processes messages from clients, updates states and potentially produces other dynamic entities such as attacks. In addition to position, which all entities have, the player also has information about how many times it has been hit and how many times it has hit others. The player character also has a message queue to receive messages from other dynamic entity. At the end of its update, it enqueues itself for the next update unless the client it represents has disconnected.

- The frontal cone attack is a one-shot task that finds player characters in its designated area and sends messages to those hit so they can update their counters, as well as back to the attacking player informing about how many were hit.

- The projectile is a short lifetime active entity that moves in the world, checks if it has hit anything and reschedules itself for another update, unless it has hit something or ran to the end of its range. The projectile can only hit one target.

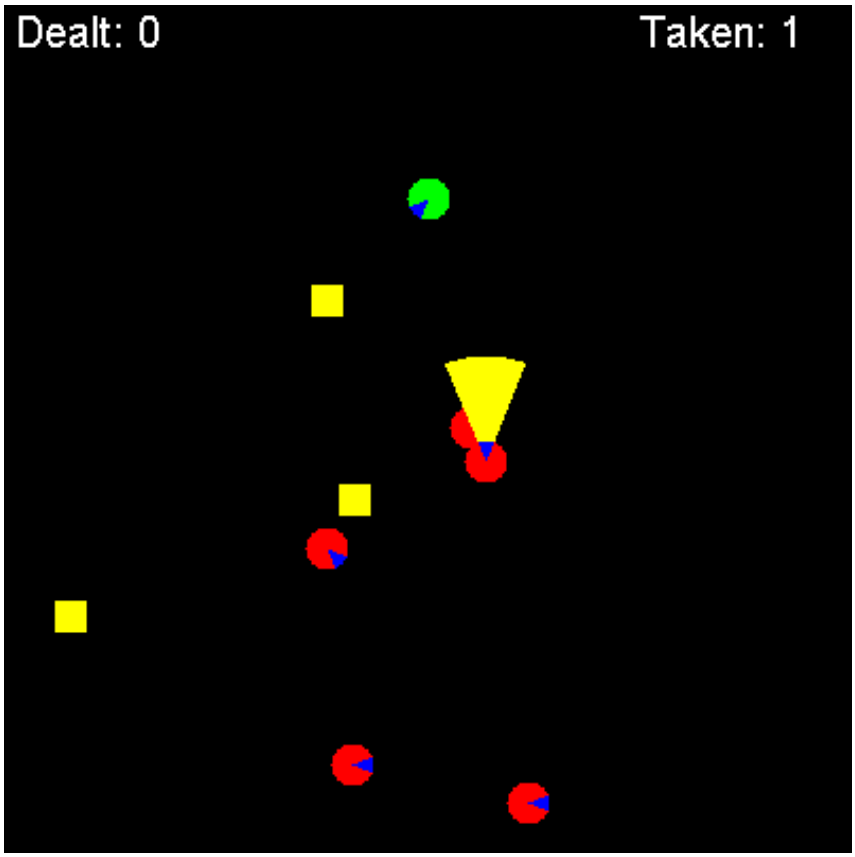


Figure 6.2: Screen shot of a game with six players. The size of the wold can be adjusted, and any number of players can join until the server is overloaded.

To simulate an MMOG workload that grows linearly with number of players, especially collision checks with the ground and other static entities, we have included a synthetic load, which emulates collision detection with a high-resolution terrain mesh. The synthetic load ensures that the cache is regularly flushed to enhance the realism of our game server prototype compared to a large-scale game server.

The game used in these experiments is simple, but it contains examples of all elements typically available in the action-based parts of a typical MMO-like game.

The system described here is implemented in Java. This programming language has strong support for multi-threading and has well-tested implementations of all the required components. The absolute values resulting from these experiments depend strongly on the complexity of the game, as a more complex game would require more processing. In addition, the absolute values depend on the runtime environment, especially the server hardware, and the choice of programming language also influence absolute results from the experiments. However, in the focus of this work are the relative results, as we are interested in comparing scalability of the multi-threaded solution with a single-threaded approach and whether the multi-threaded implementation can handle the quadratic increase in traffic as new players join and all players talk to everyone else.

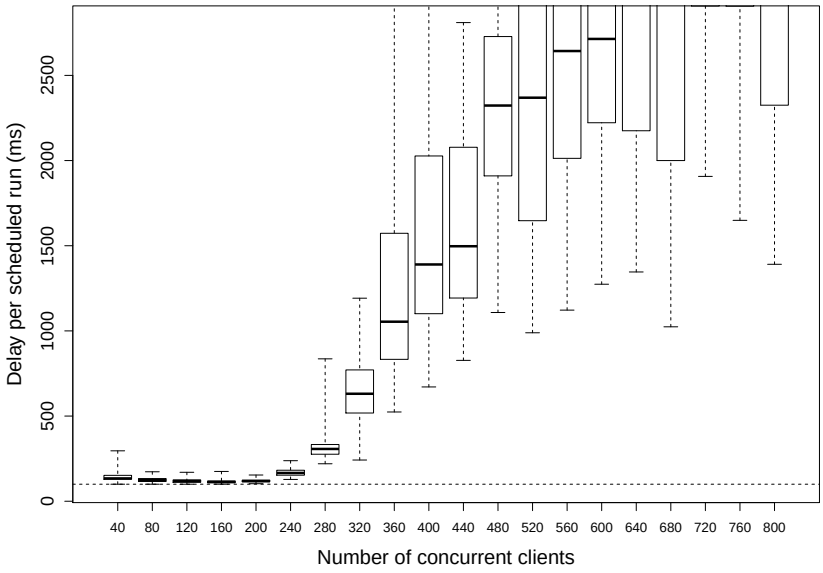
6.5 Evaluation

To have a realistic behaviour of the game clients, the game was run with 5 human players playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back enough times in parallel to simulate the desired number of clients. To ensure that client performance is not a bottleneck, the simulated clients were distributed among multiple physical machines. We monitored memory, CPU and network utilisation in these machines and made sure none of them were heavily loaded. Furthermore, as an average client generates 2.6 kbps network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing a 4 Dual-Core AMD Opteron 8218 (2600 MHz) with 16 GB RAM. To ensure comparable numbers, the server software was restarted between each test run.

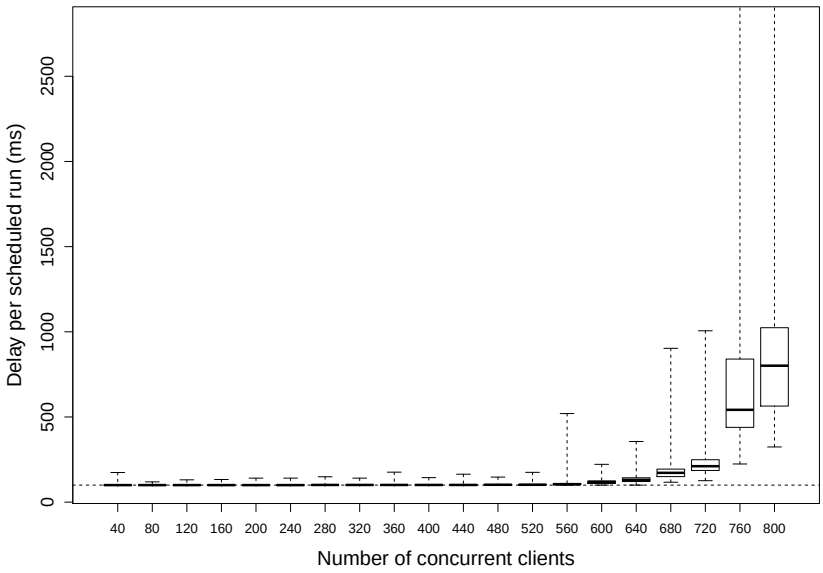
6.5.1 Response latency

The most important performance metric for client-server games is response latency from the server. From a player perspective, latency is only visible when it exceeds a certain threshold. Individual peaks in response time are obvious to the players and have the most impact on the Quality of Experience (QoOE), hence we focus on peak values as well as averages in the evaluation.

The experiments ran with client numbers ranging from 40 to 800 in increments of 40, where the goal was to keep server response time close to 100 ms. This would allow for 50 ms network latency and client response time while still responding within the 150 ms we found as a threshold for response time in section 4.5.1. We ran one experiment using a pool of 48 worker threads. We chose 48 threads, or 6 threads per core because this should be enough to keep the



(a) Single-threaded server



(b) Multi-threaded server

Figure 6.3: Response time for single- and multi-threaded servers (dotted line is the 100 ms threshold).

cores busy without excessive switching. Later, we will get back to this choice. For comparison, we ran a single-threaded version.

From figure 6.3a, we can see that the single-threaded implementation is struggling to support 280 players at an average latency close to 100 ms. The median response time was 299 ms, and it already has extreme values all the way to 860 ms, exceeding the threshold for acceptable QoE. The multi-threaded server in figure 6.3a, on the other hand, is handling the players well up to 640 players where we are getting samples above 1 second, and the median is at 149 ms.

These statistics are somewhat influenced by the fact that the number of samples is proportional to the update frequency. This means that long update cycles to a certain degree get artificially lower weight.

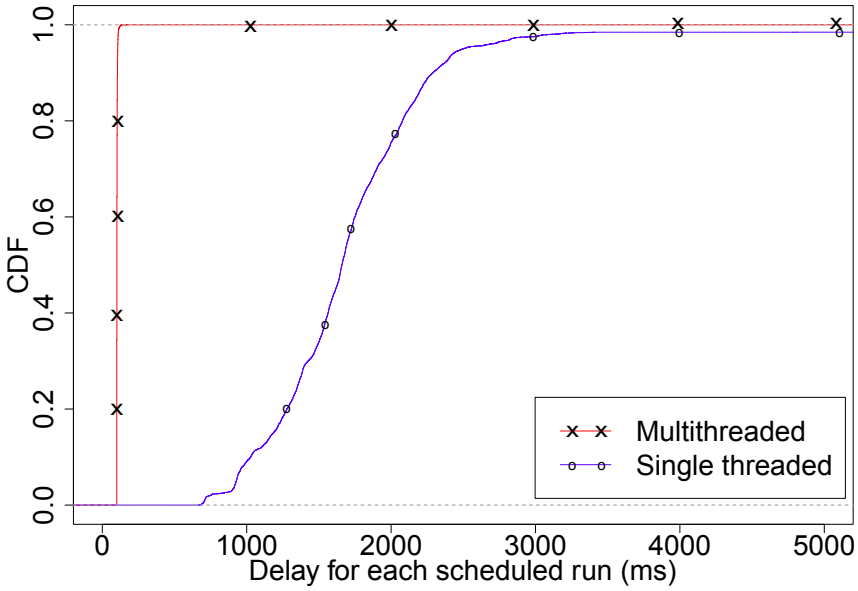
Figure 6.4 shows details of two interesting cases. In figure 6.4a, the single-threaded server is missing all its deadlines with 400 concurrent players while the multi-threaded version is processing almost everything on time. At 800 players (figure 6.4b), the outliers are going much further for both cases. Here, even the multi-threaded implementation is struggling to keep up though it is still handling the load significantly better than the single-threaded version, which is generally unplayable.

6.5.2 Resource consumption

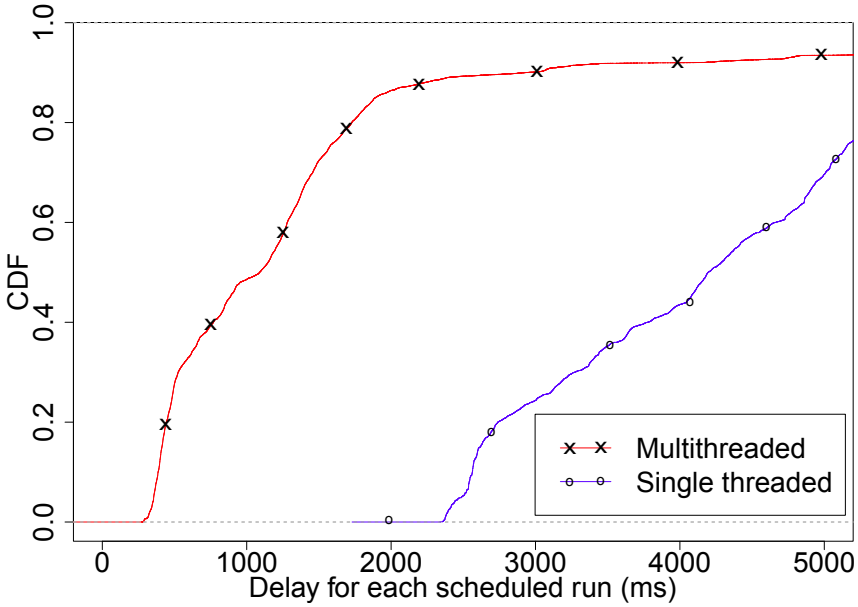
We have investigated the resource consumption when players connect, with 30 ms intervalls to the multi-threaded server as shown in figure 6.5. We present the results for 620 players, as this is the highest number of simultaneous players the server handles before any significant degradation in performance, as shown in figure 6.3b. The mean response time is 133 ms, above the ideal delay of 100 ms. Still, the server is able to keep the update rate smooth, without significant spikes. The CPU utilisation grows while the clients are logging on, and then stabilises at an almost full CPU utilisation for the rest of the run. The two spikes in response time happen while new players log in to the server at a very fast rate. Receiving a new player requires a lock in the server; hence this operation is, to a certain degree, serial.

6.5.3 Effects of thread-pool size

To investigate the effects of the number of threads in the thread-pool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. 700 clients were chosen, as this number slightly overloads the server. The number of threads in the pool was increased in increments of 2 from 2 to 256. In figure 6.6, we see clearly that the system utilises more than 4 cores efficiently, as the 4 thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency towards more consistent low response times with more available threads, to about 40 threads. In our design, the dynamic entities responsible for client avatars are responsible for communicating with their respective clients. This could mean that threads are occasionally waiting for I/O operations. Since thread-pools are not pre-emptive, such situations would lead to one core going idle if there are no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead. The results show that the average is slowly increasing after about 50 threads, though the 95-percentile is still decreasing with in-



(a) 400 concurrent clients



(b) 800 concurrent clients

Figure 6.4: CDF of response time for single- and multi-threaded servers with 400 and 800 concurrent clients.

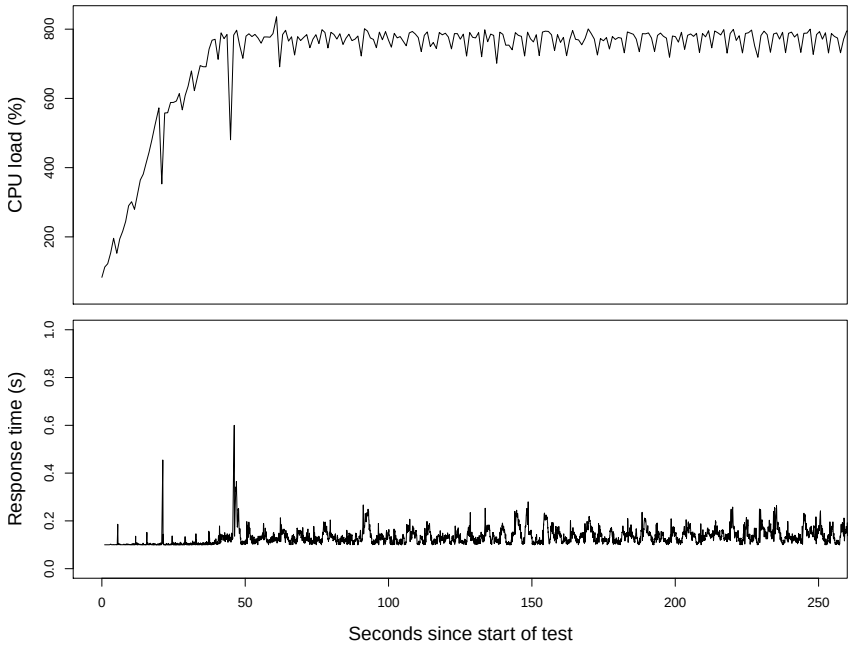


Figure 6.5: CPU load and response time over time for 620 concurrent clients on the multi-threaded server.

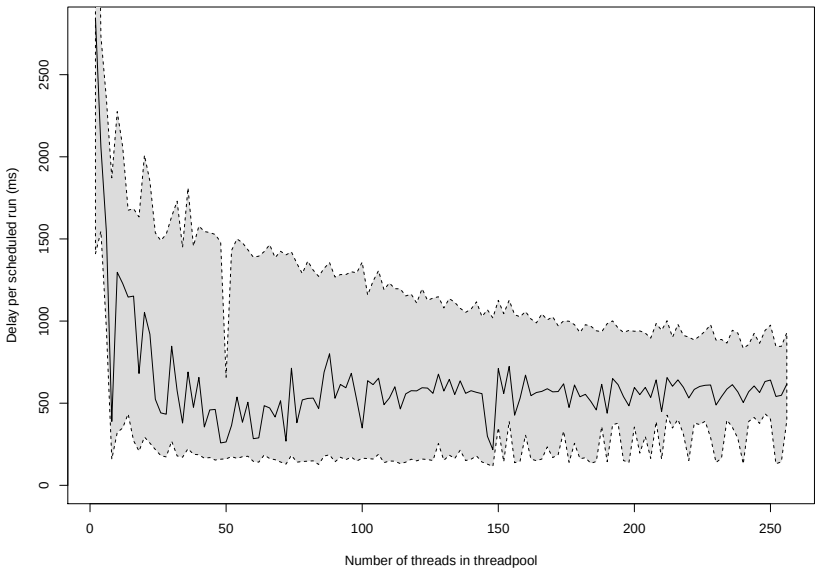


Figure 6.6: Response time for 700 concurrent clients using varying number of threads. Shaded area from 5 to 95 percentiles.

creased number of threads, up to about 100. From then on the best case is worsening again most likely due to context switching overhead.

A game developer needs to consider this trade-off when tuning the parameters for a specific game.

6.6 Discussion

Because game servers deliver entertainment, no requirement is absolute: Games can always be designed to take advantage of more computing, or restricted to require less. Competitive pressure, however, works towards heavier computations while still requiring fast response times.

Most approaches to multi-threaded game server implementations in the literature, such as Abdelkhalek and Bilas (2004b), use some form of *spatial partitioning* to lock geographical region in the game world while allowing separate regions to run in parallel. Spatial partitioning is also used in other situations to limit workload. The number of players that game designers can allow in one area in a game server is limited by the worst-case scenario. The worst-case scenario for a spatially partitioned game world is when everybody moves to the same point, where spatial partitioning still ends up with everybody in the same partition regardless of granularity. Our approach allows distribution of load independent of the distribution of the players.

We have worked to increase the player density as defined in section 2.3, or the number of players that can continually interact directly. Our approach can be combined with traditional approaches such as sharding to scale other dimensions such as total number of players. Instead of partitioning players to achieve parallelism, we relax constraints on deterministic execution order. This allows more fine-grained distribution of processing. The LEARS approach does have *limitations* and is, for example, not suitable if the outcome of a message puts restrictions on an object's state. This is mainly a game design issue, but doing full transactions can accommodate situations such as trades. Because game servers generally are backed by databases these could be used to enforce atomicity of such transactions.

Moreover, the design also adds some overhead in that the code is somewhat more complex, because all communication between elements in the system needs to go through message queues. The same issue also creates some runtime overhead, but our results still demonstrate a significant benefit in terms of the supported number of clients.

Tasks in a thread-pool can not be pre-empted, but the threads used for execution can. This distinction creates an interesting look into the performance trade-off of pre-emption. If the number of threads in the thread-pool is equal to the number of CPU cores, we have a fully co-operative multitasking system. Increasing the number of threads allows for more pre-emption, but introduces context-switching overhead.

The next logical step in scaling would be to move to multiple physical machines communicating across a network. Such an approach could give us even more processing power for each player, allowing more detailed simulation. This could be applied to more detailed physics or more complex artificial intelligence. However, because our design generally does not partition any way, all threads have an equal probability of communicating with any other thread. Thus, if the server was distributed across a network, it is possible that the communication overhead would overwhelm any gains in performance from added processing power. The effectiveness

of such a design depends on how much communication there is relative to computation. If each player is computationally expensive, a server cluster architecture could be viable. Evaluating the merits of such an approach requires further experiments. Otherwise, a hybrid approach would be ideal. An architecture similar to LEARS could be used to increase player density scaling, while traditional approaches, such as spatial partitioning, could be employed to utilise multiple machines.

Further, our absolute numbers are difficult to relate to real games. Computing power required for each client varies widely between games. We have shown that using the LEARS design can increase the scalability of a game server, but exactly how effective it is depends on the game and the system running the server.

Our design seems best suited for games where interaction between a large number of players at the same time is important. At the same time, only relatively fast-paced games can utilise the increased availability of computing power for each client. From these criteria, traditional MMO-RPGs as well as MMO-FPSes, as defined in section 1.1 are ideally suited.

6.7 Summary

This chapter has looked at a novel approach to building parallel game servers. Game servers are an interesting workload with an otherwise rare combination of requirements. Because this approach requires a complete redesign of the game server architecture, we made our own proof-of-concept game server that achieves improved scaling by relaxing ordering constraints. We found that this server allowed good scaling of player density with number of threads. However, our experiment used a simplified experimental system benchmark, and more work is required to determine how the system performs in a fully realistic setting.

Chapter 7

Paper and author's contributions

The PhD period started with a motivation to work on aspects of delay in games. My papers reflect a large variation in approaches to this central theme. Because the approaches are so varied, I cooperated with many different people who brought a diverse set of skills and knowledge to the research. One paper (Raaen and Grønli, 2014) describes a survey of existing work on how delay affects gameplay. The next two papers (Raaen and Petlund, 2015; Raaen and Kjellmo, 2015) look at user interface delays in different types of gaming setups. Further, we present two papers (Raaen et al., 2014; Raaen and Eg, 2015) on how users perceive delay in user interfaces. Then, we include one paper (Raaen et al., 2014) measuring how cloud infrastructure can be utilised for game servers. The last paper (Raaen et al., 2012) propose a novel approach to parallelising workloads of a typical game server. Because work on the different questions proceeded in parallel, the papers are presented in a logical order rather then chronologically.

7.1 Paper I: Latency thresholds for usability in games: A survey

Kjetil Raaen and Tor-Morten Grønli

Abstract User interactions in interactive applications are time critical operations; late response will degrade the experience. Sensitivity to delay does however vary greatly with between games. This paper surveys existing literature on the specifics of this limitation. We find a classification where games are grouped with others of roughly the same requirements. In addition we find some numbers on how long latency is acceptable. These numbers are however inconsistent between studies, indicating inconsistent methodology or insufficient classification of games and interactions. To improve classification, we suggest some changes. In general, research is too sparse to draw any strong or statistically significant conclusions. In some of the most time critical games, latency seems to degrade the experience at about 50 ms.

Lessons learned In this paper, we worked through existing material on how delay affects games. We discovered that almost all existing work focused on network latency in client-server games. We also found that very few authors had looked for correlations between

the skill of the player and effects of delay. Further, different authors use different classification for games, making it unclear how existing conclusions apply to new games.

Author's contributions Raaen did the bulk of the work on this paper, both gathering data and writing. The co-author provided guidance regarding methodology and collaborated on the writing.

Published in Norsk Informatikkonferanse (NIK) 2015

ISSN 1892-0713

7.2 Paper II: How much delay is there really in current games?

Kjetil Raaen and Andreas Petlund

Abstract All computer games present some delay between human input and results being displayed on the screen, even when no networking is involved. A well-balanced discussion of delay-tolerance levels in computer games requires an understanding of how much delay is added locally, as well as in the network. This demonstration uses a typical gaming setup wired to an oscilloscope to show how long the total, local delay is. Participants may also bring their own computers and games so they can measure delays in the games or other software.

Results show that local delays constitute such a large share of the total delay that that it should be considered when studying the effects of delay in games, often far exceeding the network delay evaluated.

Lessons learned For this paper, we designed and run an experiment to determine exact delay from user input to screen output in games. We found that local delays in gaming systems can be significant compared to network delays. Because earlier work has ignored local delay, this finding significantly changes how we interpret their results. It also shows that we need to be careful to include local delay in our own work when examining how humans handle delay.

Author's contributions Raaen designed, built and conducted the experiment. Co-authors contributed to writing, but here, too Raaen had overall responsibility.

Published in ACM Multimedia Systems (MMSys), ACM 2015

DOI 10.1145/2713168.2713188

7.3 Paper III: Measuring Latency in Virtual Reality Systems

Kjetil Raaen and Ivar Kjellmo

Abstract Virtual Reality (VR) systems have the potential to revolutionise how we interact with computers. However motion sickness and discomfort are currently severely impeding

the adoption. Traditionally the focus of optimising VR systems have been on frame-rate. Delay and frame-rate are however not equivalent. Latency may occur in several steps in image processing, and a frame-rate measure only picks up some of them. We have made an experimental setup to physically measure the actual delay from the user moves the head until the screen of the VR device is updated. Our results show that while dedicated VR-equipment had very low delay, smartphones are in general not ready for VR-applications.

Lessons learned For this paper, we designed and ran an experiment to measure exact time delay between rotating a VR headset and the screen updating. We found that dedicated virtual reality hardware had very low delays. Conversely, the current trend of using smartphones as virtual reality displays seems premature, because current phones respond slowly.

Author's contributions Raaen designed the physical side of the experiment, while the co-authors designed the virtual setup. We gathered the data in close cooperation and split the work on writing between the two authors, with Raaen taking final responsibility.

Published in International Conference on Entertainment Computing 2015, Springer 2015

DOI 10.1007/978-3-319-24589-8_40

7.4 Paper IV: Can gamers detect cloud delay?

Kjetil Raaen, Ragnhild Eg and Carsten Griwodz

Abstract In many games, a win or a loss is not only contingent on the speedy reaction of the players, but also on how fast the game can react to them. From our ongoing project, we aim at establishing perceptual thresholds for visual delays that follow user actions. In this first user study, we eliminated the complexities of a real game and asked participants to adjust the delay between the push of a button and a simple visual presentation. At the most sensitive, our findings reveal that some perceive delays below 40 ms. However, the median threshold suggests that motor-visual delays are more likely than not to go undetected below 51-90 ms. These results can in future investigations be compared to thresholds for more complex visual stimuli, and to thresholds established from different experimental approaches.

Lessons learned This is our first experiment seeking to measure human sensitivity to delays. First, we see that this sensitivity is highly individual. Next, we find no correlation between gaming experience and sensitivity to delays. We have later reevaluated the findings in this paper in light of new data, but the general conclusions hold.

Author's contributions This paper is the result of cooperation between Eg and Raaen. All the way from designing the experiment to the writing we worked together. Raaen implemented the experiment software and wrote the parts of background related to gaming, while Eg designed the protocol and wrote the background from cognitive psychology. Griwodz consulted and worked on the analysis and text.

Published in The 13th Annual Workshop on Network and Systems Support for Games (NetGames), ACM 2014

DOI 10.1109/NetGames.2014.7008962

7.5 Paper V: Instantaneous human-computer interactions: Button causes and screen effects

Kjetil Raaen and Ragnhild Eg

Abstract Many human-computer interactions are highly time-dependent, which means that an effect should follow a cause without delay. In this work, we explore how much time can pass between a cause and its effect without jeopardising the subjective perception of instantaneity. We ran two experiments that involve the same simple interaction: A click of a button causes a spinning disc to change its direction of rotation, following a variable delay. In our adjustment experiment, we asked participants to adjust the delay directly, but without numerical references, using repeated attempts to achieve a value as close to zero as possible. In the discrimination task, participants made judgements on whether the single rotation change happened immediately following the button-click, or after a delay. The derived thresholds revealed a marked difference between the two experimental approaches, participants could adjust delays down to a median of 40 ms, whereas the discrimination mid-point corresponded to 148 ms. This difference could possibly be an artefact of separate strategies adapted by participants for the two tasks. Alternatively, repeated presentations may make people more sensitive to delays, or provide them with additional information to base their judgements on. In either case, we have found that humans are capable of perceiving very short temporal delays, and these empirical results provide useful guidelines for future designs of time-critical interactions.

Lessons learned Here we changed the nature of the stimulus presented in the previous paper, using a moving stimulus. We also extend the experiment to include a new mechanism to record perceived delays. This new mechanism allows only one exposure to a given delay level before being asked if it was delayed. We found a large discrepancy between the mechanisms, and hypothesised that multiple exposures to delay makes it easier to detect.

Author's contributions This is the second product of the cooperation between Eg and Raaen. Again we worked together from designing the experiment to writing. Raaen implemented the experiment software and wrote the parts of background related to gaming, while Eg designed the protocol and wrote the background from cognitive psychology.

Published in International Conference on Human-Computer Interaction 2015, Springer 2015

DOI 10.1007/978-3-319-21006-3_47

7.6 Paper VI: Is todays public cloud suited to deploy hard-core realtime services?

Kjetil Raaen, Andreas Petlund and Pål Halvorsen

Abstract *Cloud computing* is a popular way for application providers to obtain a flexible server and network infrastructure. Providers deploying applications with tight response time requirements such as games, are reluctant to use clouds. An important reason is the lack of real-time guarantees. This paper evaluates the actual, practical soft real-time CPU performance of current cloud services, with a special focus on online games. To perform this evaluation, we created a small benchmark and calibrated it to take a few milliseconds to run (often referred to as a microbenchmrak). Repeating this benchmark at a high frequency gives an overview of available resources over time. From the experimental results, we find that public cloud services deliver performance mostly within the requirements of popular online games, where Microsoft Azure Virtual machines give a significantly more stable performance than Amazon EC2.

Lessons learned This paper looked at cloud computing services and investigated if their performance characteristics are suitable for running workloads typical of game servers. Somewhat surprisingly, we found highly fluctuating performance in the cloud services we tested. Some servers seemed to stall for long enough that it would be noticeable for players. From this, we decided not to go on and design game server architectures specialised for cloud distribution.

Author's contributions Raaen gathered data and wrote much of the text in this paper. The coauthors contributed writing and evaluation.

Published in Second Workshop on Large Scale Distributed Virtual Environments on Clouds and P2P (LSDVE) – Euro-Par 2013: Parallel Processing Workshops, Springer, 2013

DOI 10.1007/978-3-642-54420-0_34

7.7 Paper VII: LEARS: A Lockless, relaxed-atomicity state model for parallel execution of a game server partition

Kjetil Raaen, Havard Espeland, Hakon K. Stensland, Andreas Petlund, Pal Halvorsen and Carsten Griwodz

Abstract Supporting thousands of interacting players in a virtual world poses huge challenges with respect to processing. Existing work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players needs to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency or inconsistency. It is a major challenge of game engines to scale such areas to the largest

number of players possible; in a deviation from earlier thinking, parallelism on multi-core architectures is applied to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that need timely updates. We also measure how the response time for the multi-threaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game-server can be an embarrassingly parallel problem.

Lessons learned We searched for better ways of utilising parallelism for keeping response-times low with increasing numbers of players in massively multiplayer games. Our solution involves relaxing some requirements of atomicity and ordering. Because games are in nature not deterministic, it is not critical that the server is deterministic. We found that by relaxing these requirements we could improve parallelism in a simulated server environment.

Author's contributions Raaen brought the original idea and implemented the experiment software. He also led and contributed to the writing process with the other authors.

Published in Proceedings of the International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS) – The 2012 International Conference on Parallel Processing Workshops, IEEE, 2012.

DOI 10.1109/ICPPW.2012.55

7.8 Other Publications

The author contributed to several other publications during the PhD period that did not fit into the scope of this thesis. Instead we provide a short summary of each.

Demonstrating Hundreds of AIs in One Scene An extension of the work presented in paper VII, this demo shows how the LEARS architecture handles AI loads, specifically pathfinding (Raaen and Petlund, 2013).

Games for Research: A Comparative Study of Open Source Game Projects Going through literature on how to optimise game infrastructure, we find that most use either purpose-built prototypes or work on games that are not popular (Halvorsen and Raaen, 2014a). The main problem here is finding a game that is representative of popular games and also open source so it can be modified to test the hypothesis the researcher is working on. As part of his Master Thesis Stig Halvorsen created a list of requirements and a list of potential games, and together we wrote a paper on the results.

Implementing Virtual Clients in Quake III Arena As part of his master thesis Stig Halvorsen created a system for load testing a game server with realistic loads (Halvorsen and Raaen, 2014b). To achieve this, he moved the code for computer controlled opponents in the

open source game *Quake 3 Arena* to the clients. This allows the server to see the computer controlled players as real clients and respond accordingly. This setup allows researchers working on networks or server infrastructure to create loads corresponding to any number of players without actual people. While Halvorsen did most of the technical work, we cooperated on writing the paper.

Chapter 8

Conclusion

This thesis looks for answers to the questions described in section 1.2 about latency in games. Solving this problem requires answers to multiple questions. We identified some of the important questions and have worked towards answers. Each question required a different approach, but they all come together to illuminate the main question.

8.1 Summary

How much delay is there really in current games? Question 1 from section 1.2 is the focus of chapter 3. Suspecting that measurements of delay in software do not capture the full delay, we decided to use a measurement device external to the system being measured. This allows us to measure total delay from an input is sent to the system until the results are visible on screen. We performed two such experiments, one where the input was a mouse click, and another where the input was a rotation of a virtual reality display. We did not measure a large number of systems and software, but those few we do measure show large differences.

How fast should game systems respond to avoid deterioration of player experience? In chapter 4, we looked into question 2 from section 1.2. We found that measuring a player's sensitivity to motor-visual delays is not a straightforward endeavour. We approached the matter by asking our participants to evaluate if a computer's reactions immediately followed their actions. They manipulated the delay directly, but without feedback, in an experiment designed to assess human sensitivity to motor-visual delays, over a series of trials. The experiment included several conditions that also explored possible interacting factors, visual stimulus size and delay, as well as previous gaming experience. Due to the repetitive nature of the experiment, we saw that the assigned task challenged the patience of some participants. In turn, our results initially showed a large influence from those who did not take the task seriously and accepted far higher delay values than the majority.

This approach is limited by the simple nature of the interaction as well as only investigating if the users are aware of a delay.

Are cloud services responsive enough to run game servers? To investigate question 3 from section1.2, we measured delays inherent in public cloud services and compared them

to acceptable delays. In chapter 5, we used a microbenchmark to measure the stability of CPU performance in three service classes from each of two providers. While not a comprehensive list of services, these results give us an indication of the state of cloud computing for games. It seems that performance stability is not consistent enough for fast-paced games at the moment.

How can parallelism be better utilised in game servers for massive multiplayer games? In chapter 6, we suggest one possible appraoch solving to question 4 from section 1.2. Firstly, we identified a clear distinction between scaling absolute number of player and scaling player density, and decided to focus on the latter. We have shown that we can improve resource utilisation by distributing load across multiple CPUs in a unified memory multi-processor system. This distribution is made possible by relaxing constraints to the ordering and atomicity of events. The system scales well, even in the case where all players must be aware of all other players and their actions. The thread pool system balances load well between the cores, and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other.

8.2 Main contributions

While working on these questions, we find various individual results that contribute to answering the overall question, but also bring interesting results on their own.

We suggest a simple and reliable method to measure actual delays in typical gaming setups. This setup is independent of both hardware and software and only requires access to the initial signal from the user to the system. Using this setup allows developers and researchers to accurately measure total delay in their interactive systems. Our results show that internal measures of delay are insufficient, and that peripheral devices add significant delay. For the mouse click, delays vary from 60 to 170 ms in the tested setups, but can also be reduced to 30 ms by sacrificing visual quality. The virtual reality systems designed for this purpose were extremely fast, with the best reacting in less than 5 ms. Improvised virtual reality solutions based on smartphones are at the moment too slow for the purpose.

It seems that awareness of total delay is limited both in scientific work and in the computer games industry, except for the specialised case of virtual reality. Often, frame-rate is used as a proxy for delay. Local delay is rarely reported in works on delay in general, possibly because it is considered insignificant in comparison to the values measured, such as network latency. While players and game journalists to some degree seem aware of these delays, scientific work on delay in games does in general not take this into account.

In virtual reality interactions, delays are better understood. Various methods to measure these delays have been proposed previously. Our solution is simpler and easier to use. We also provide measurements on some modern virtual reality systems, showing that dedicated systems have delays of 4 to 63 ms. Smartphones, sometimes used as cheap alternative VR-displays on the other hand have delays of 46 to 96 ms.

With these conclusions in mind, we present work on how much delay can be present in a system before users notice. This question turns out to be more complicated than it seems

on the surface. Going through existing work, we find two different fields that touch upon the subject. Work specifically on games tends to ignore local delays. Experimental conditions were also very difficult to replicate. The field of psychophysics does take local delay into account and often describe repeatable experiment conditions, but their experiments are difficult to relate directly to games.

To evaluate sensitivity to delay in scenarios closer to games, we conduct an experiment on actual people. We let participants trigger actions and apply various delays to this interaction. Measurements rely on participants' self-reported experience of delay. From this, we notice large individual variations. This means that testing on a few people is not enough to know that delays will go unnoticed. Further, we have indications that numbers of repetitions influence detection thresholds, with more repetitions giving lower thresholds. Many users can consistently detect delays around 150 ms, and there is a long tail of users who seem to notice even shorter delays. This means there is a significant overlap between delays detectable by users, and delays actually present in many systems today.

Having established some benchmarks, we investigate how stable CPU response is in cloud services. CPU stalls usually translate to delay that would be added by hosting game servers in public clouds. It seems the conclusions about CPU usage from (Barker and Shenoy, 2010) still hold for Amazon EC2. We find that the cheap instance from Amazon EC2 are not good enough ($c_v = 0.45$), but it is better at the higher service levels ($c_v = 0.039$ and $c_v = 0.19$ respectively). Variance at all levels is larger than for Azure. An important element to keep in mind for Amazon EC2 is that migrating to a different instance is necessary if a change in service level is needed. For the more time sensitive class of games, these delays can add up with network latency to unacceptable levels. Even for less sensitive applications there are some caveats. We do get rare samples where the CPU seems to have stalled for multiple seconds, which could discourage users.

Microsoft Azure Virtual machines seem to give more stable performance than Amazon EC2, without the extreme latency peaks ($c_v = 0.050$, $c_v = 0.048$ and $c_v = 0.047$ respectively). Utilising these results can allow providers of interactive online services to be more agile and react faster to changes in demand. Although it gives less processor speed for the same price, it is more reliable for planning stability, reducing the need for a specialised service for managing changes between service levels. If a change in service level is needed, Azure allows for dynamic changes without the need to migrate the virtual machine. Infrastructure of cloud providers changes fast; our numbers might already be outdated. However, awareness of potential performance stability in cloud services is important to developers who consider deploying their real-time products to the cloud.

To improve server processing delays, we chose to work on server backends for massive multiplayer games. Games can scale in many dimensions. For massive multiplayer games, one of the most important dimensions is player density: The number of players that can fit into a given area of the game world and interact. Such interactions create a complex computational problem. By making a simplified game utilising parallelism, we attempted to improve the state of the art. A significant difficulty in parallel games is synchronisation. When multiple things are happening at the same time, all entities that might potentially be affected must be locked to ensure complete consistency. By taking advantage of the fact that ordering in games is arbitrary anyway, we were able to relax some consistency constraints and increasing parallelism. Thus,

we are able to challenge the traditional sequential game loop.

Implementing this system and running on a powerful multi-CPU system can allow many hundreds of players gathering in the same location before server response rates become unacceptable. Our results indicate that it is possible to design an *embarrassingly parallel* game server. We also observe that the implementation is able to handle a quadratic increase of in-server communication when many players interact in a game-world hotspot. Scaling game servers is a varied task. We have focused on improving the density of players supported. For improvements in metrics such as world size and total number of players, our solution should be combined with other, improvements such as spatial partitioning.

8.3 Future work

Because this work has a wide focus, it presents a whole range of new opportunities for research. Each area we have worked on has produced its own unique questions.

From the work on total delay in games, it is clear that delays are much longer than those under the control of the programmer. Which components add this delay? Are there ways to minimise this delay? Some of this information might be available in documentation for the hardware. However, because this documentation is often lacking, and there are many trade secrets in the pipeline, answers might be hidden in blackboxes, and thus need to be answered empirically by conducting more experiments and measurements.

Our work on virtual reality systems shows that some of these provide very fast response, and earlier work has shown that delay might be a factor in motion sickness triggered by virtual reality. Further empirical work is needed to conclude how long delays can be without degrading user experience. An investigation into which other factors contribute to cybersickness and how to alleviate this would be interesting.

We found some answers on the impact of delay on human-computer interactions, but much work is still left. So far, we have worked on general limits for sensitivity to delays, finding that this sensitivity varies greatly between individuals. In the future, we would like to look for patterns in who are the most and least sensitive individuals. Can we find something to predict sensitivity to delays? Because games are very different in how fast reactions are required to play the game effectively, we need different values for different games. Thus, we would like to find reasonable estimates for acceptable delay in various types of games.

By starting with the very simplest interactions, we are not sure how the complexities of real interactions affect results. Conversely, traditional work has started with real games, including all their randomness, and multiple things happening at the same time. Finding a middle ground between the two methods, with simplified but not trivial games, would give opportunities for more detailed and general conclusions.

Our work on responsiveness of cloud services is already two years old, and these services change fast. A new look would be useful. More service providers should also be included, as well as metrics for input-output operations.

The extension of game server architectures could also be taken further. Our work so far used a very simplified game as a proof of concept. Ideally, the approach should be implemented in a full, complete massive multiplayer game. This should give results that are more realistic, at least

with respect to this specific game. To investigate the performance of such an implementation supporting thousands of players, computer controlled players, *bots* could be used (Halvorsen and Raaen, 2014b).

The work also focuses only on a single, shared memory machine. A good goal would be to move on to distributed architectures. This could be achieved by implementing cross-server communication directly in the server code, or by using existing technology that makes cluster behave like shared memory machines.

8.4 Applying our conclusions in practice

Going back to the original question, we now have at least some answers.

- Make sure to measure total delay in the game. This metric will be an important quality metric for a game. Do not trust internal measurements of such delay. Values above 150 ms will be consciously noticed by many players. If the game includes fast paced action elements, consider aiming for even faster response times to appease the most sensitive individuals and account for potential subconscious effects of delay.
- Be aware of the potential performance stability issues in virtualised cloud servers. For a response-time critical application clouds might not provide the stable, fast response-times needed.
- Make highly parallel code for the server, and carefully evaluate all consistency requirements. Remember that the application is a game. Strict ordering of events might not be necessary. Position information does not need to be strictly consistent. Any such relaxation can give large performance bonuses.

Maybe if we, in the future, keep these and many other results in mind when making games, future players can only blame themselves if their avatars are eaten by spiders.

Bibliography

Abdelkhalek, A. and A. Bilas (2004a). Parallelization and performance of interactive multiplayer game servers. *In Proceedings 18th International Parallel and Distributed Processing Symposium*, 72–81.

Abdelkhalek, A. and A. Bilas (2004b). Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 72.

Amazon (2013). Amazon ec2 instance types. http://aws.amazon.com/ec2/instance-types/.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *American Federation of Information Processing Societies Conference Proceedings*.

Amin, R., F. Jackson, J. Gilbert, J. Martin, and T. Shaw (2013). Assessing the impact of latency and jitter on the perceived quality of call of duty modern warfare 2. In M. Kurosu (Ed.), *Human-Computer Interaction. Users and Contexts of Use*, Volume 8006 of *Lecture Notes in Computer Science*, pp. 97–106. Springer Berlin Heidelberg.

Apple (2015). Apple thunderbolt display. https://www.apple.com/displays/specs.html.

Armitage, G. (2003). An experimental estimation of latency sensitivity in multiplayer quake 3. In *The 11th IEEE International Conference on Networks 2003 ICON2003*, pp. 137–141. IEEE.

Asus (2015). Asus rog swift pg278q. http://www.asus.com/Monitors/ROG_SWIFT_PG278Q/specifications/.

Barker, S. K. and P. Shenoy (2010). Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, New York, NY, USA, pp. 35–46. ACM.

Beigbeder, T., R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool (2004). The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of 3rd Workshop on Network and system support for games (NetGames)*, NetGames '04, New York, NY, USA, pp. 144–151. ACM.

Bernier, Y. (2001). Latency compensating methods in client/server in-game protocol design and optimization. *Game Developers Conference*.

Beskow, P. B., G. A. Erikstad, P. Halvorsen, and C. Griwodz (2009). Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 10:1—-10:6.

Bethesda Softworks (1994). Elder scrolls.

Blizzard Entertainment (1996). Diablo.

Blizzard Entertainment (1998). Starcraft.

Blizzard Entertainment (2004). World of warcraft.

BlurBusters (2014). Preview of nvidia g-sync. http://www.blurbusters.com/gsync/preview2/.

BlurBusters (2015). List of 120hz monitors includes 144hz, 240hz. http://www.blurbusters.com/faq/120hz-monitors/.

Chen, K.-T., Y.-C. Chang, and P.-H. Tseng (2011). Measuring the latency of cloud gaming systems. *MM '11 Proceedings of the 19th ACM international conference on Multimedia*, 1269–1272.

Chen, K.-t., P. Huang, G.-s. Wang, C.-y. Huang, and C.-l. Lei (2006). On the sensitivity of online game playing time to network qos. *Proceedings of IEEE INFOCOM 00*(c).

Chen, K.-T. and C.-L. Lei (2006). Network game design: Hints and implications of player interaction. In *Proceedings of 5th Workshop on Network and System Support for Games (NetGames)*, NetGames '06, New York, NY, USA. ACM.

Chen, K.-T. and C.-L. Lei (2012). Are all games equally cloud-gaming-friendly? an electromyographic approach. *11th Annual Workshop on Network and Systems Support for Games (NetGames)*, 1–6.

Chu, H. S. (2008). Building a simple yet powerful mmo game architecture. http://www.ibm.com/developerworks/architecture/library/ar-powerup1/.

Claypool, M. (2005). The effect of latency on user performance in real-time strategy games. *Computer Networks 49*(1), 52–70.

Claypool, M. and K. Claypool (2006). Latency and player interaction in online games. *Communications of the ACM 49*(11), 40–45.

Claypool, M. and K. Claypool (2010). Latency can kill : Precision and deadline in online games. *Proceedings of the First ACM Multimedia Systems Conference*.

Claypool, M. and D. Finkel (2014). The effects of latency on player performance in cloud-based games. *13th Annual Workshop on Network and Systems Support for Games (NetGames)*.

Cunningham, D. W., V. A. Billock, and B. H. Tsou (2001). Sensorimotor adaptation to violations of temporal contiguity. *Psychological Science 12*(6), 532–535.

Dally, B. (2009). The end of denial architecture and the rise of throughput computing. *Keynote speech at Desgin Automation Conference*.

Davis, S., K. Nesbitt, and E. Nalivaiko (2014). A systematic review of cybersickness. In *Proceedings of the Conference on Interactive Entertainment*, New York, NY, USA. ACM.

Dell (2015). Dell ultrasharp 24 ultra hd monitor. http://accessories.us.dell.com/sna/productdetail.aspx?sku=860-BBCD.

Denning, P. J., D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young (1989). Computing as a discipline. *Communications of the ACM 32*(1), 9–23.

Di Luca, M. (2010). New method to measure end-to-end delay of virtual reality. *Presence: Teleoperators and Virtual Environments 19*(6), 569–584.

Dick, M., O. Wellnitz, and L. Wolf (2005). Analysis of factors affecting players' performance and perception in multiplayer games. In *Proceedings of 4th Workshop on Network and System Support for Games (NetGames)*, NetGames '05, New York, NY, USA, pp. 1–7. ACM.

Ebert, J. P. and D. M. Wegner (2010). Time warp: authorship shapes the perceived timing of actions and events. *Consciousness and Cognition 19*(1), 481–489.

Egenfeldt-Nielsen, S., J. H. Smith, and S. P. Tosca (2009). *Understanding Video Games: The Essential Introduction*. Taylor & Francis.

El-Khamra, Y., H. Kim, S. Jha, and M. Parashar (2010). Exploring the performance fluctuations of hpc workloads on clouds. *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, 383–387.

Electronic Arts (1994). Need for speed.

Electronic Arts (1995). Command & conquer.

Electronic Arts (1999). Medal of honor.

Epic Games (1999). Unreal tournament.

Foster, I., Y. Zhao, I. Raicu, and S. Lu (2008). Cloud computing and grid computing 360-degree compared. *Grid Computing Environments Workshop*, 1–10.

Friedman, M. (1937). The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association 32*(200), pp. 675–701.

Fritsch, T., H. Ritter, and J. Schiller (2005). The effect of latency and network limitations on mmorpgs: a field study of everquest2. *Proceedings of 4th workshop on Network and system support for games (NetGames)*.

Fujisaki, W. and S. Nishida (2009). Audiotactile superiority over visuotactile and audiovisual combinations in the temporal resolution of synchrony perception. *Experimental Brain Research 198*(2-3), 245–259.

Gajinov, V., O. S. Unsal, E. Ayguad, and T. Harris (2009). Atomic quake: using transactional memory in an interactive multiplayer game server. *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 25–34.

Gescheider, G. A. (1976). *Psychophysics: The Fundamentals* (3rd ed.). Psychology Press.

Goldsmith, J. T. T. and M. E. Ray (1948). Cathode-ray tube amusement device.

Haggard, P. and V. Chambon (2012). Sense of agency. *Current Biology 22*(10), R390—-R392.

Halvorsen, S. and K. Raaen (2014a). Games for research: A comparative study of open source game projects. In *Euro-Par Parallel Processing Workshops*, Volume 8374 of *Lecture Notes in Computer Science*, pp. 353–362. Springer Berlin Heidelberg.

Halvorsen, S. M. and K. Raaen (2014b). Implementing virtual clients in quake iii arena. In *Entertainment Computing-ICEC 2014: 13th International Conference*, Volume 8770, pp. 232. Springer.

Hayes, D. A., I.-J. Tsang, D. Ros, A. Petlund, and B. Briscoe (2015). Internet latency: Causes, solutions and trade-offs. In *EuCNC Special session on latency*.

Henderson, T. (2001). Latency and user behaviour on a multiplayer game server. In J. Crowcroft and M. Hofmann (Eds.), *Networked Group Communication*, Volume 2233 of *Lecture Notes in Computer Science*, pp. 1–13. Springer Berlin Heidelberg.

Heron, J., J. V. M. Hanson, and D. Whitaker (2009). Effect before cause: Supramodal recalibration of sensorimotor timing. *PLoS ONE 4*(11), e7681.

International Telecommunication Union (ITU-T) (2014). Methods for the subjective assessment of small impairments in audio systems bs series. *Recommendation BS.1116-2 2*.

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley.

Jarschel, M., D. Schlosser, S. Scheuring, and T. Hoßfeld (2011). An evaluation of qoe in cloud gaming based on subjective tests. *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 330–335.

Jota, R., A. Ng, P. Dietz, and D. Wigdor (2013). How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Paris, pp. 2291–2300.

Kijima, R. and T. Ojika (2002). Reflex hmd to compensate lag and correction of derivative deformation. *Proceedings IEEE Virtual Reality*.

Lee, K., D. Chu, E. Cuervo, A. Wolman, and J. Flinn (2014). Outatime: using speculation to enable low-latency continuous interaction for mobile cloud gaming. *Proceedings of the 12th annual international conference on Mobile systems, applications, and services - MobiSys '14*, 347–347.

Lewis, J., P. Trinh, and D. Kirsh (2011). A corpus analysis of strategy video game play in starcraft: Brood war. *Proceedings of the 33rd annual conference of the cognitive science society.*, 687–692.

March, S. T. and G. F. Smith (1995). Design and natural science research on information technology. *Decision Support Systems 15*(4), 251–266.

Mennie, D. (1976). Consumer electronics electronic gamesmanship. *Spectrum, IEEE 13*(December), 27–30.

MicroProse, Activision, Infogrames Entertainment, and 2K Games (1991). Civilization.

Microsoft (2013). Microsoft azure pricing details. ttp://www.windowsazure.com/en-us/pricing/details/.

Microsoft Studios (2005). Forza motorsport.

Moore, G. (1965). Cramming more components onto integrated circuits. *Electronics 38*(8).

Müller, J. and S. Gorlatch (2007). Enhancing online computer games for grids. In V. Malyshkin (Ed.), *Parallel Computing Technologies*, Volume 4671 of *Lecture Notes in Computer Science*, pp. 80–95. Springer Berlin / Heidelberg.

Nichols, J. and M. Claypool (2004). The effects of latency on online madden nfl football. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '04, New York, NY, USA, pp. 146–151. ACM.

Nvidia (2015). Nvidia grid. http://shield.nvidia.com/grid-game-streaming.

Occelli, V., C. Spence, and M. Zampini (2011). Audiotactile interactions in temporal perception. *Psychonomic Bulletin & Review 18*(3), 429–454.

Olson, J. L., D. M. Krum, E. a. Suma, and M. Bolas (2011). A design for a smartphone-based head mounted display. *Proceedings - IEEE Virtual Reality*, 233–234.

Ostermann, S., A. Iosup, and N. Yigitbasi (2010). A performance analysis of ec2 cloud computing services for scientific computing. *Cloud Computing*, 115–131.

Pantel, L. and L. C. Wolf (2002). On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '02, New York, NY, USA, pp. 23–29. ACM.

Petlund, A. (2009). *Improving latency for interactive, thin-stream applications over reliable transport*. Phd thesis, Simula Research Laboratory / University of Oslo, Unipub, Oslo, Norway.

Platt, P. A. (1990). *Real-Time Flight Simulation and Head-Mounted Display - An Inexpensive Approach to Military Pilot Training*. Ph. D. thesis, Air Force Institute of Technology.

Pourazad, M., C. Doutre, M. Azimi, and P. Nasiopoulos (2012). Hevc: The new gold standard for video compression: How does hevc compare with h.264/avc? *IEEE Consumer Electronics Magazine 1*(3), 36–46.

Quax, P., P. Monsieurs, W. Lamotte, D. De Vleeschauwer, and N. Degrande (2004). Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd Workshop on Network and system support for games (NetGames)*, NetGames '04, New York, NY, USA, pp. 152–156. ACM.

Raaen, K. and R. Eg (2015). Instantaneous human-computer interactions : Button causes and screen effects. *International Conference on Human-Computer Interaction 2015*, 1–12.

Raaen, K., R. Eg, and C. Griwodz (2014). Can gamers detect cloud delay ? *2014 13th Annual Workshop on Network and Systems Support for Games (NetGames) 200*, 7–9.

Raaen, K., H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz (2012). Lears: A lockless, relaxed-atomicity state model for parallel execution of a game server partition. In *2012 41st International Conference on Parallel Processing Workshops*, pp. 382–389. IEEE.

Raaen, K. and T.-M. Grønli (2014). Latency thresholds for usability in games : A survey. *Norsk Informatikkonferanse*.

Raaen, K. and I. Kjellmo (2015). Measuring latency in virtual reality systems. *International Conference on Entertainment Computing*, 1–6.

Raaen, K. and A. Petlund (2013). Demonstrating hundreds of ais in one scene. *Entertainment Computing ICEC 2013*, 195–199.

Raaen, K. and A. Petlund (2015). How much delay is there really in current games? *Proceedings of the 6th ACM Multimedia Systems Conference*, 2–5.

Raaen, K., A. Petlund, and P. Halvorsen (2014). Is today's public cloud suited to deploy hardcore realtime services? In *Euro-Par: Parallel Processing Workshops*, Lecture Notes in Computer Science. Springer Berlin Heidelberg.

Renderingpipeline (2013). Measuring input latency. http://renderingpipeline.com/2013/09/measuring-input-latency/.

Rohde, M., M. Scheller, and M. O. Ernst (2014). Effects can precede their cause in the sense of agency. *Neuropsychologia 65*, 191–196.

Schad, J., J. Dittrich, and J. Quiané-Ruiz (2010). Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment 3*(1).

Seow, S. C. (2008). *Designing and Engineering Time*. Addison-Wesley.

Sheldon, N., E. Girard, S. Borg, M. Claypool, and E. Agu (2003). The effect of latency on user performance in warcraft iii. In *Proceedings of the 2nd Workshop on Network and System Support for Games (NetGames)*, NetGames '03, New York, NY, USA, pp. 3–14. ACM.

Stuckel, D. and C. Gutwin (2008). The effects of local lag on tightly-coupled interaction in distributed groupware.

Swindells, C., J. Dill, and K. Booth (2000). System lag tests for augmented and virtual environments. *Proceedings of the 13th annual ACM symposium on User interface software and technology. 2*, 161–170.

Vaquero, L. M., L. Rodero-merino, J. Caceres, and M. Lindner (2009). A break in the clouds : Towards a cloud definition. *Computer Communication Review 39*(1), 50–55.

Waldo, J. (2008). Scaling in games and virtual worlds. *Commun. ACM 51*(8), 38–44.

Wegner, P. (1995). Interaction as a basis for empirical computer science. *ACM Computing Surveys 27*(1), 45–48.

White, W., A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan (2007). Scaling games to epic proportions. *Proceedings of the ACM SIGMOD international conference on Management of data*, 31.

White, W., B. Sowell, J. Gehrke, and A. Demers (2008). Declarative processing for computer games. In *Proceedings of the ACM SIGGRAPH symposium on Video games*, pp. 23–30.

Wilcoxon, F. (1946). Individual comparisons of grouped data by ranking methods. *Journal of economic entomology 39*(6), 269.

Yahyavi, A. and B. Kemme (2013). Peer-to-peer architectures for massively multiplayer online games. *ACM Computing Surveys 46*(1), 1–51.

Part II

Research Papers

Paper I

Latency thresholds for usability in games: A survey

Latency Thresholds for Usability in Games: A Survey

Kjetil Raaen
Westerdals Oslo School of Art, Communication and Technology
Faculty of Technology
University of Oslo, Institute for Informatics
Simula Research Laboratory

Tor-Morten Grønli
Westerdals Oslo School of Art, Communication and Technology
Faculty of Technology

Abstract

User interactions in interactive applications are time critical operations; late response will degrade the experience. Sensitivity to delay does however vary greatly with between games. This paper surveys existing literature on the specifics of this limitation. We find a classification where games are grouped with others of roughly the same requirements. In addition we find some numbers on how long latency is acceptable. These numbers are however inconsistent between studies, indicating inconsistent methodology or insufficient classification of games and interactions. To improve classification, we suggest some changes.

In general, research is too sparse to draw any strong or statistically significant conclusions. In some of the most time critical games, latency seems to degrade the experience at about 50 ms.

1 Introduction

Whenever a human interacts with a computer, the computer could be said to run an "interactive application". A user enters some input and the computer responds. Word processors, spreadsheets and web-browsers are based on a workflow where the user continuously enters input, and the system responds immediately. Conversely, not all applications have this interaction as its central function. Simulations running on supercomputers spend very little time interacting with the user, and most of its time is spent doing calculations; these applications are often said to do "batch processing".

Games are a class of applications that commonly requires significant amounts of relatively fast interactions. They are also among the few applications where users regularly measure and worry about latency since this has a high impact on gameplay.

This paper was presented at the NIK-2015 conference; see http://www.nik.no/.

This make games one of the most interesting categories of applications for studies on delays.

Research has often focused on response time as a critical factor for quality of experience (QoE) in networked games. This metric is equally relevant for local games, but achieving sufficient response time is rarely a problem in this class of applications. The dominating factor of response time in networked games and interactive worlds are usually network latency; hence earlier work is mostly focused on network metrics. Conversely in local games, lacking any communication, response time is only bound by computation, which is a known factor when creating the game and can thus be compensated for.

Data on acceptable latency in games, or on how player performance correlates with latency is sparse in current literature. In some cases multiple papers cite the same few sources of empirical data.

Work on improving latency of games often quote 100ms as acceptable latency for fast-paced games, for example Raaen et al. 2012 [19]. Other types of games are quoted with more lax latency requirements. Our goal is finding if this is reasonable, or if there are other, better estimates. Games are very different in how fast reactions are necessary to play the game effectively. It is reasonable to assume that the response time they require depends on the speed of gameplay. The primary goal of this paper is to investigate the field of latency in games through a survey of existing literature. This will contribute to defining latency thresholds from a theoretical perspective and can further be used in research projects aiming to improve interactive applications.

2 Background

In any interactive application, there will be a delay between the time the user sends input and the result appearing on screen. This delay is commonly termed ”input lag”. If the computer communicates through a network, any communication over this network takes time, called ”network latency” or simply ”latency”. Most players typically use the term ”lag” for both types of delay. This can make complaints from users somewhat ambiguous.

Types of Games

”Latency and player interaction in online games” [8] by Mark Claypool and Kajal Claypool 2006), describes a whole set of game types and situations and recommend latency limits, dividing games into three broad categories:

First Person Avatar Describes games where the player controls an avatar, and the game is displayed from the point of view of that avatar, as if the player sees ”through the eyes” of the avatar. The most common variant of this class of game is the *First Person Shooter* (FPS)

Third Person Avatar These games are also based on a single avatar controlled by the player, but the avatar is seen from the outside. The most studied variant in this group is the *Massive Multiplayer Online*(MMO) game.

Omnipresent In these games, the player does not control a specific avatar, rather multiple elements in a large game world. The most popular variant of this is the *Real Time Strategy* (RTS) game.

It is assumed that each of these classes of games has different latency requirements. The MMO genre is a particularly interesting case, and the games themselves as well as the technology behind them have received significant attention from computer scientists.

> One of the main attractions of these games lies in the number of players that participate in the game. The more players that are in the game world, the more interactive, complex and attractive the game environment will become. [22]

Though MMOs are technically the most technologically demanding, Claypool does not place them in the most latency sensitive category.

Cloud Gaming vs. Client-Server

client-server games are all designed from the ground up to handle network latency, deviation or displacement in phase timing and jitter (understood as irregular variation). By employing various prediction techniques and allowing a looser consistency in state between different players, these games will be able to alleviate, or in some cases completely isolate, the players from the effects of latency. Some of the most ubiquitous techniques are described in Bernier [4].

Initially, this paper describes the basic client-server case, where the client transfers data to the server; the server does necessary processing and sends back the result, which the client renders. However, strict adherence to the client-server pattern is rare except in purely experimental games. By allowing the client to do some calculations locally, feedback can be much quicker. Usually referred to as *Client Side prediction*, this solution will, at its simplest, run exactly the same code on the client as would run on the server. Whenever an update is received from the server, client state is reverted to conform to this data. Furthermore, instant effects, such as weapons firing, are executed immediately on the local client to give the player a feeling of responsiveness. To allow players to hit where they aim, the system needs to predict where other players are at a given moment, increasing complexity of the system even further.

The emerging concept of cloud gaming represents ”Software as a Service” [21] where the software is a game. Delivering games this way presents some unique challenges, possibly including different requirements on the network latency. Latency in a cloud gaming scenario will work differently than latency in a traditional client-server game. Here, the client more closely resembles a traditional ”thin client”, which simply forwards commands from the user to the server, and displays video from the server. The cloud, or remote server, performs all game logic and rendering tasks. In this scenario, requirements on client hardware will be very low. As tradeoff, the requirements on the network link are significantly stricter. There are at least two clear reasons for this. First, transferring high definition video requires significantly higher throughput than simple control signals; secondly, none of the techniques to alleviate network delay described above can be applied. Evaluations on these requirements are scarce at the moment.

3 Findings

The goals of this study is to investigate the issue of latency in games based on previous work and identify potential avenues for further empirical research on this topic. In order to do this we surveyed the area, searching IEEE, ACM and Springer databases, and selected papers were the title, keywords and abstract implied that the research focused on acceptable latency limits. Further, we traced the actual data presented in each paper to its original sources and included this paper. Filtering of relevant papers were based on the criterion on empirical data about how players react to latency in games. Table 1 list main contributors to the area. These are further presented in the rest of this section.

Controlled Studies

The oldest paper in this survey [12] dates back to 2001. Running a FPS game-server open to the Internet, the authors observe player behaviour and how this is affected by network delay. Most players connecting to their server fall into the interval 50 ms to 300 ms. Beyond this their only result relevant here is that players with delays over 400 ms seem much more likely to leave immediately.

Panthel and Wolf [17] study racing games, claiming this to be the most sensitive class of game. Using a setup with two identical machines, with controlled latency between them, the authors run two very different experiments. First they set up an identical starting position and perform identical actions on both sides, observing discrepancies. In the games studied, this leads to both players seeing themselves in the lead at the same time, even at the lowest tested latency of 100 ms. Next, they allow players to play actual games under varying latencies to the server. They find that the *average* player's performance deteriorates first, at their lowest tested latency of 50 ms. The beginners drive too slowly to notice this delay, while excellent players are able to compensate for more latency. Performance of the excellent drivers degrade sharply at 150 ms.

Perhaps the most cited paper in this study is "Latency and player interaction in online games" [8] by Claypool and Claypool. Most authors citing this paper simply uses it as an explanation for setting acceptable limits to response times from the system they are evaluating. The value 100ms is frequently quoted. Among these are [6] which we discuss elsewhere in the literature review.

However, Claypool's work and conclusions are much more nuanced, categorising games based on how they interact with the player. The paper describes a whole set of game types and situations and recommend latency limits. These limits are estimated at 100ms for "first person avatar" games, 500ms for "third person avatar" games, and 1000ms for "omnipresent" games. Further, the authors analyse different actions within each type of game. For each of these items, they have used empirical studies showing how player performance varies with network latency. The commonly quoted 100ms figure is based on multiple types of actions in a first person avatar game, the most latency-sensitive class of games according to this paper. It is important to notice that 100ms represents a point where player performance already has dropped off sharply from the previous data point 75ms. Because systems should be designed to withstand worst-case scenarios, the design goal should be below 75 ms. This paper is however mostly a secondary source, citing data from earlier work.

Though the 100ms estimate for "first person avatar" games is usually given with reference to [8], that study actually cites results from a paper by Beigbeder et al.

Table 1: Papers included in this study, the type of game studied and type of study.

Authors/Year	Title	Game Type	Study Type
Henderson 2001 [12]	Latency and user behaviour on a multiplayer game server.	FPS	semi-controlled experiment
Panthel and Wolf 2002 [17]	On the Impact of Delay on Real-time Multiplayer Games	Racing Game[1]	controlled experiment
Armitage 2003 [2]	An experimental estimation of latency sensitivity in multiplayer Quake 3.	FPS	observational study
Sheldon et al. 2003 [20]	The effect of latency on user performance in Warcraft III	RTS	controlled experiment
Beigbeder et al. 2004 [3]	The effects of loss and latency on user performance in unreal tournament 2003.	FPS	controlled experiment
Nicholas and Claypool 2004 [16]	The Effects of Latency on Online Madden NFL Football	Sports game[2]	controlled experiment
Quax et al. 2004 [18]	Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game.	FPS	controlled experiment
Fritch et al. 2005 [11]	The Effect of Latency and Network Limitations on MMORPGs	various	semi-controlled experiment
Dick 2005 [10]	Analysis of factors affecting players' performance and perception in multiplayer games.	MMO	observational study, controlled experiment
Claypool 2005 [7]	The effect of latency on user performance in Real-Time Strategy games	RTS	controlled experiment
Chen et al. 2006 [5]	On the Sensitivity of Online Game Playing Time to Network QoS.	MMO	observational study
Claypool and Claypool 2010 [9]	Latency Can Kill: Precision and Deadline in Online Games.	various	controlled experiment
Jarschel et al. 2011 [14]	An Evaluation of QoE in Cloud Gaming Based on Subjective Tests.	Cloud Gaming	controlled experiment
Amin et al. 2013 [1]	Assessing the impact of latency and jitter on the perceived quality of call of duty modern warfare 2	FPS	controlled experiment

[1] Racing games can be categorised as both first and third person avatar games.
[2] Sports games are difficult to classify in Claypool's model. *Madden NFL Football* can be classified as third person avatar or omnipresent.

2004 [3], and the 2006 paper adds no new data, only analysis. Players are set in front of computers running the game "Unreal Tournament 2003" and given a set of tasks. The widely quoted number seems to originate in an experiment where shooting precision was tested. The experiment was run three times by two different players at each latency level. Results from such an experiment are not sufficient to draw any clear conclusions. Other factors were also analysed in this work, but are of little statistical relevance due to the extremely low number of participants. Other numbers cited in [8] are similarly from studies using an extremely low number of participants.

For "omnipresent" games, the most relevant data comes from "The effect of latency on user performance in Real-Time Strategy games". Claypool [7], which in turn uses most of the data from Sheldon et al. [20]. The papers do not establish any clear threshold for latency in such games, simply concluding that 1000ms should be completely safe. Splitting the games in different types of interaction receives significant focus, but differences in skill levels of players are completely ignored. Only two players participate in either study. For the last category of game, third person avatar, the data comes from Fritch et al. [11], which evaluates the performance of two players playing the game *Everquest 2* under different conditions.

In "Latency Can Kill: Precision and Deadline in Online Games" [9], Claypool further elaborates on the categories of games and actions. Each action is described by two parameters; deadline and precision. Deadline is the time an action takes to complete, and precision is the accuracy needed by the player. To investigate how sensitivity to latency varies with these two parameters, the authors modified a game, Battle Zone capture the flag (BZFlag), so these parameters could be controlled directly. Each scenario was then played out using computer controlled player avatars called *bots*. They do not clearly justify that bots are an accurate model for how human players react to latency, neither do they cite any research indicating that this is the case. The hypothesis of a correlation between each of the variables and latency sensitivity was supported, but not strongly.

Team sports games are played in a somewhat different manner to the others mentioned here. Usually you have control of one character at a time, as in a third person avatar game, but you switch character often, depending on who is most involved in the action, as if in an omnipresent game. Nichols and Claypool [16] study the sports game *Madden NFL Football*, and conclude that latencies as high as 500 ms are not noticeable.

In [18], the authors set up a 12 player match of *Unreal Tournament 2003* in a controlled environment. Each player is assigned a specific amount of latency and jitter for the duration of the match. After the match, the players answer a questionnaire about their experience in the game. This study still uses relatively few players, but they are able to conclude that 60ms of latency noticeably reduces both performance and experience of this game. In contrast to [5], [18] finds no effects of jitter. The most probable explanation for this is that jitter is handled well in this game.

Amin et al. [1] also run controlled experiments, but use subjective measures on the FPS game "Call of Duty Modern Warfare 2". Further, they graded participants according to gaming experience. They conclude that the most experienced users are not satisfied with latencies above 100 ms.

Observational Studies

Others have taken different approaches to determining acceptable latency for games. Armitage [2] set up two different servers for the game Quake 3 and monitored the latencies experienced by the players joining the servers. They assume that players will only join servers which have acceptable latency. This methodology might give insight as to how much latency players think is acceptable. However, the study does not take into account which other servers are available, so the results will be heavily influenced by the presence of lower latency servers or lack thereof. Additionally, the latencies players think are acceptable might not be the same as the limit where their performance degrades.

Another interesting approach is that of Chen et al. [5]. They examine an online RPG, *ShenZhou Online*. By Claypool's classification this game wold be a third person avatar game, and hence be less sensitive to latency than the games discussed earlier. Instead of using a controlled lab environment the authors chose to analyse network traces from an existing, running game. They asked the question: "Does network QoS influence the duration of play sessions?" The Quality of Service (QoS) factors they examined were packet loss, latency and jitter. They hypothesis was that if the underlying network conditions affect the players negatively, it should show up in the players' enjoyment of the game, and hence their motivation to keep playing.

Between 45 and 75 ms RTT, the authors find a linear correlation between increased latency and decreased game session length. For standard deviation of latency, which would represent "jitter", the correlation is even stronger. At extreme RTT values or extreme jitter the trend is reversed though, and the authors surmise that there is a group of players who are used to bad connections, and another group of players who keep the game on while not really paying attention. These results indicate negative impact of latencies much lower than the 100ms mentioned in earlier literature. Session length as indicator for player satisfaction is an ingenious approach, but the chain of effect from network latency to session length is complicated, and there is significant room for hidden variables.

Dick et al. [10] uses two separate methods to investigate the question. International Telecommunication Union defines what they call an "impairment scale" [13], which defines a rating called " Using the Mean Opinion Score (MOS) [13] ratings, they ran an online survey asking people how much delay would cause each level of impairment. Players reported they could play "unpaired" at up to about 80 ms, and "tolerable" at 120 ms for most games. Further, the authors ran a controlled experiment testing different latencies. Their results show large differences between games, with the most sensitive game "Need for Speed Underground 2" showing impairment even at the lowest tested delay of 50 ms.

Cloud Gaming

Latency in cloud gaming is much less studied than for networked games. Jarschel et al.2011 [14] test players' subjective experience of varying network latencies and amount of packet loss in cloud gaming. Using a setup where the gameplay was transferred over a network mimicking the cloud scenario, the authors introduced the varying Quality of Service (QoS) parameters and asked their 48 participants how they liked the service in each scenario. They concluded that a latency of 80ms was noticeable in the fastest-paced game. In all games, packet loss in the stream from server to client was extremely detrimental to the experience. However, their

conclusion can only be used to put an upper bound on latency sensitivity in cloud gaming, because they conducted no experiments between latencies of 0ms and 80ms.

4 Discussion

Based on our review of the field and the findings retrieved from primary sources, we further discuss the implications and general ideas that can be gained from current research.

Limitations in the Classification

The game classification in [8] does not reflect the full spectrum of games, as games in each type can have significantly different latency requirements. In the research summarised in this article, conclusions about noticeable latency in third person avatar games range from 45 ms [5] to over 500 ms [8]. Different games in the same category have very different modes of interaction. Third person avatar games can have very different ways of interaction with the environment. Some are 3D graphical representations of very abstract game mechanics, where relative positions and timing are almost irrelevant, while others are highly detailed simulations of realistic physics where small changes in position and timing make for large changes in the outcome of actions.

For first person avatar games, the differences are often in playing style. Some games focus on tactical movement and cover (for example *Metal Gear Solid*), while others rely purely on reaction time and precision motor skills (for example *Unreal Tournament*). It is likely that the last category has stricter latency requirement than the first. No studies found in this survey address these questions.

The distinction between these types of games is also blurred by the fact that many games allow multiple points of view. Role playing games and racing games commonly support both first person and third person view modes for the same game; leaving it up to the user to decide.

Even the status of omnipresent games is not entirely clear. Some are clearly slow-paced and highly strategic, but not all. The well known game *Star Craft* is one of the most popular games in this category. The papers cited here all seem to agree that this means it should not have strict latency requirements. However, this game is played as a professional sport in parts of the world. In these matches players take around 400 individual actions per minute. This equates to 150 ms per action [15]. A hypothesis that players playing on this level require latencies at least lower than the time between actions would be interesting to investigate.

Types of Input

Users control games in very different ways. FPS and RPG games are sometimes controlled with a handheld controller using analogue joysticks and buttons and in other cases with a keyboard and mouse. Many games have support for both types of input. None of the articles we have surveyed mention which type of input is used, though it is more than likely that this affects the sensitivity of the game. Other games may even have specialised control hardware, such as steering wheels for racing games or digital, or arcade style, joysticks for retro games.

Skill Level

Different players might have different requirements; more skilled players will probably notice latency earlier than amateurs, because they will be more aware of details of the game. On the other hand, experienced players might also have more experience in compensating for and dealing with latency while playing. Players who are used to low latency might react sooner than those who usually play over high-latency links and are used to this situation. None of the studies discussed so far supply strong support for or reject such hypothesis.

Player Assumptions

Most games report latency to players during playing, and all allow for checking this information. Only one source [10] attempts to compare player expectations to their actual experience in games, and finds indications that they are very different. Among the other studies, some partly measure player expectation, such as those that measure session length and connection numbers to a running server. Others attempt measuring the actual effects of latency, but none make it clear how they avoid introducing effects of player expectation.

5 Conclusions and Future Work

This paper has investigated acceptable latency for games. Current research is mostly inconclusive about latency requirements for networked games. In general, it seems that 60 ms [18], or even 45 ms [5] are better estimates at how much latency is acceptable in the most fast-paced games than the traditionally quoted 100ms value. Furthermore, there are no clear, consistent results available and the diversity in game scenarios make comparisons challenging. Studies suffer from uncontrolled environments or very limited numbers of participants. Studies using both a controlled environment *and* a number of participants that is large enough to do statistical analysis would do a lot to clarify the situation.

Towards a New Classification

Current classification of games is based on genre distinctions used in game design and criticism. These distinctions do not map neatly to what we need to distinguish on a technical level. Considering the limitations of the current classification of games, it is clear that from a technical perspective an updated classification is required. Some factors of this classification present themselves from the work surveyed here.

- *Spatial precision* describes how much precision in input affects the outcome of an action. This can be the exact angle of the joystick or the distance moved on a mouse.
- *Temporal precision* describes how much timing of user actions affects the outcome of an action.
- *Input type* describes how users interact with the game, both in terms of physical devices in use and how these devices control gameplay.

These parameters are not immediately obvious from looking at the game, but they are objective results of the game code. Further research is required to establish testing protocols for these parameters, as well as further elaboration of the model.

Cloud Gaming

Even in the most studied case of networked applications, conclusions are highly diverging. Cloud gaming, the newer case, has barely been studied at all. Current work also lack input from important fields such as neuropsychology in designing the investigations. However, the topic of latency in games is relevant and will influence potentially all games utilising some version of network-based interaction. Therefore we suggest this area is worthy of future pursuit. Forthcoming research would strongly benefit from being based in concrete prototype implementations, in order to generate datasets for more precise conclusions.

References

[1] Amin, R., Jackson, F., Gilbert, J., Martin, J., and Shaw, T. Assessing the Impact of Latency and Jitter on the Perceived Quality of Call of Duty Modern Warfare 2. In *Human-Computer Interaction. Users and Contexts of Use*, M. Kurosu, Ed., vol. 8006 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 97–106.

[2] Armitage, G. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *The 11th IEEE International Conference on Networks 2003 ICON2003* (2003), IEEE, pp. 137–141.

[3] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E., and Claypool, M. The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2004), NetGames '04, ACM, pp. 144–151.

[4] Bernier, Y. Latency compensating methods in client/server in-game protocol design and optimization. *Game Developers Conference 98033*, 425 (2001).

[5] Chen, K.-t., Huang, P., Wang, G.-s., Huang, C.-y., and Lei, C.-l. On the Sensitivity of Online Game Playing Time to Network QoS. *Proceedings of IEEE INFOCOM 2006 00*, c (2006).

[6] Choy, S., Wong, B., Simon, G., and Rosenberg, C. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)* (Nov. 2012), 1–6.

[7] Claypool, M. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks 49*, 1 (Sept. 2005), 52–70.

[8] Claypool, M., and Claypool, K. Latency and player interaction in online games. 40–45.

[9] Claypool, M., and Claypool, K. Latency Can Kill : Precision and Deadline in Online Games. *Proceedings of the First ACM Multimedia Systems Conference* (2010).

[10] Dick, M., Wellnitz, O., and Wolf, L. Analysis of Factors Affecting Players' Performance and Perception in Multiplayer Games. In *Proceedings of 4th ACM SIGCOMM Workshop on Network and System Support for Games* (New York, NY, USA, 2005), NetGames '05, ACM, pp. 1–7.

[11] Fritsch, T., Ritter, H., and Schiller, J. The effect of latency and network limitations on MMORPGs: a field study of everquest2. *... 4th ACM SIGCOMM workshop on Network ...* (2005).

[12] Henderson, T. Latency and User Behaviour on a Multiplayer Game Server. In *Networked Group Communication*, J. Crowcroft and M. Hofmann, Eds., vol. 2233 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2001, pp. 1–13.

[13] International Telecommunication Union (ITU-T). Methods for the subjective assessment of small impairments in audio systems BS Series. *Recommendation BS.1116-2 2* (2014).

[14] Jarschel, M., Schlosser, D., Scheuring, S., and Hoss feld, T. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (June 2011), 330–335.

[15] Lewis, J., Trinh, P., and Kirsh, D. A corpus analysis of strategy video game play in starcraft: Brood war. *... of the 33rd annual conference of the ...* (2011), 687–692.

[16] Nichols, J., and Claypool, M. The Effects of Latency on Online Madden NFL Football. In *Proceedings of the 14th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2004), NOSSDAV '04, ACM, pp. 146–151.

[17] Pantel, L., and Wolf, L. C. On the Impact of Delay on Real-time Multiplayer Games. In *Proceedings of the 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video* (New York, NY, USA, 2002), NOSSDAV '02, ACM, pp. 23–29.

[18] Quax, P., Monsieurs, P., Lamotte, W., De Vleeschauwer, D., and Degrande, N. Objective and subjective evaluation of the influence of small amounts of delay and jitter on a recent first person shooter game. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2004), NetGames '04, ACM, pp. 152–156.

[19] Raaen, K., Espeland, H., Stensland, H. K., Petlund, A., Halvorsen, P., and Griwodz, C. LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition. *2012 41st International Conference on Parallel Processing Workshops* (Sept. 2012), 382–389.

[20] Sheldon, N., Girard, E., Borg, S., Claypool, M., and Agu, E. The Effect of Latency on User Performance in Warcraft III. In *Proceedings of the 2Nd Workshop on Network and System Support for Games* (New York, NY, USA, 2003), NetGames '03, ACM, pp. 3–14.

[21] Vaquero, L. M., Rodero-merino, L., Caceres, J., and Lindner, M. A Break in the Clouds : Towards a Cloud Definition. *Computer Communication Review 39*, 1 (2009), 50–55.

[22] Yahyavi, A., and Kemme, B. Peer-to-peer architectures for massively multiplayer online games. *ACM Computing Surveys 46*, 1 (Oct. 2013), 1–51.

Paper II

How much delay is there really in current games?

How Much Delay Is There Really in Current Games?

Kjetil Raaen
Westerdals — Oslo School of Arts, Communication and Technology
1325 Lysaker, Norway
Simula Research Laboratory
University of Oslo
raakje@westerdals.no

Andreas Petlund
Simula Research Laboratory
1325 Lysaker, Norway
apetlund@simula.no

ABSTRACT

All computer games present some delay between human input and results being displayed on the screen, even when no networking is involved. A well-balanced discussion of delay-tolerance levels in computer games requires an understanding of how much delay is added locally, as well as in the network. This demonstration uses a typical gaming setup wired to an oscilloscope to show how long the total, local delay is. Participants may also bring their own computers and games so they can measure delays in the games or other software.

Results show that local delays constitute such a large share of the total delay that that it should be considered when studying the effects of delay in games, often far exceeding the network delay evaluated.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Benchmarking

General Terms

Measurement, Performance

Keywords

Games, interaction, delay, lag

1. INTRODUCTION

Researchers have worked on how delay influences players in networked games for some years now [1, 3, 5, 8, 10, 14]. This work has traditionally focused on pure network delay in multiplayer games. In these games, latency is somewhat alleviated by locally echoing user actions without waiting for server confirmation, as well as dead reckoning of other players postitions [6]. Recently there has been a trend towards studying delays in cloud gaming [11,13]. These delays are somewhat different, because they occur directly between the user's input and the observed output. No techniques are currently implemented to hide this latency.

What is lacking from from all these publications is a consideration of the local system delay. All of these studies report network latency. The network latency studied may be from actual network conditions or artificially induced. What is missing from the latency analysis are reports on how much *local delay* is present in the test setup. Local delay describes the time from a users sends an input until the result is visible on screen, for actions that do not need to traverse a network. Discussions on the influence of local latency for the player tolerance and experience is also missing. Such local latency can constitute a large part of the total latency and vary considerably depending on the hardware used and software configurations. Therefore, knowing what share the network and local delays constitute is important for a proper analysis of the effect of one of the components on user experience.

The goal of this work is to investigate and demonstrate how much total system lag is present in current computer systems used for gaming. We will allow participants to measure delays in their own hardware and software. Further, we propose a method for measuring such delay in experiments seeking to quantify individual components of gaming delay. Lastly, we will compare this delay to a similar setup using a cloud gaming service instead of local rendering.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

http://dx.doi.org/10.1145/2713168.2713188
MMSys '15, Mar 18-20, 2015, Portland, OR, USA
Copyright 2015 ACM 978-1-4503-3351-1/15/03 ...$15.00.

2. IMPORTANT COMPONENTS IN LOCAL DELAY

Even in a local game with no networking, there is potential for delays at levels perceptible to humans, or affecting human task performance. This section discusses the parts of the pipeline that add most to the response delay in the local system. Components in the pipeline from user input to screen respones are to a large degree *black boxes*; documentation about how they work is often lacking. Thus, the only way to evaluate these delays is by measuring.

2.1 Monitors

Screens used to display output add some delay, which can be divided into two parts.

2.1.1 Monitor update

LCD screens receive and display updated images at a fixed rate. This rate is termed *screen refresh* rate. Most modern screens update at 60 frames per second (FPS), or every 16.7 ms. Some screens specialised for gaming or other response critical applications can update much faster.

Table 1: Some monitors and their response time.

Brand	Type	Response time
Apple	Thunderbolt Display	12 ms [2]
Dell	UltraSharp 24	8 ms [12]
Asus	ROG SWIFT PG278Q	1 ms [4]

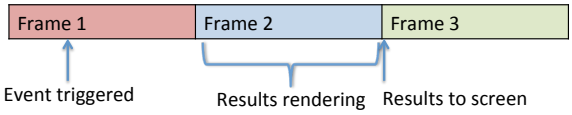


Figure 1: Timeline for events with double buffering.

2.1.2 *Monitor response time*

Monitors vary wildly in response time, that is the time they take from one shade of grey to another, from 1 ms for screens designed for gaming, to 12 ms for typical office screens, see table 1 for some examples. The exact procedure and color values for this test are not standardised, and it is reasonable to assume that manufacturers choose the conditions most favorable to their product.

2.2 Frame buffers

A frame buffer is memory used for holding a rendered frame about to be displayed on the monitor. Modern games use at least two frame buffers. To avoid showing unfinished frames on the screen, drawing happens in one buffer, while the other is being displayed. This practice is termed *double buffering*. Further, to avoid showing parts of two different buffers, it is common to wait for the next screen refresh before swapping. The terms *vertical synchronization* or *vsynch* are used, because historically the swap was performed when the electron beam was returned to the start of the frame, a period of time called the *vertical blanking interval*.

When double buffering is used, rendering follows the sequence: An event from an input device is registered during frame N, frame N+1 contains the result, and at the time of frame N+2 the result is sent to screen, as shown in figure 1. This gives a minimum of 1 full frame time from input event to result on screen. At 60 FPS this adds up to a total of 17 - 33 ms delay. Many games have a target framerate of only 30 FPS. At this rate, the delay from screen refresh and frame buffer pipeline is 33 - 67 ms. Further, not all hardware is capable of keeping up with the target framerate at all times. Slow hardware will lead to significantly longer delays.

An increased number of frame buffers in the pipeline will increase this delay, because more steps are added between rendering and displaying data. High system load from the game itself or external tasks can lead to lower frame-rate, and thus add to this number.

2.3 Input device latency

The best gaming equipment has tailor-made drivers that have been tweaked for low latency performance. A good gaming mouse may add ∼2 ms [15] to the latency. Devices that are not made for gaming may add more.

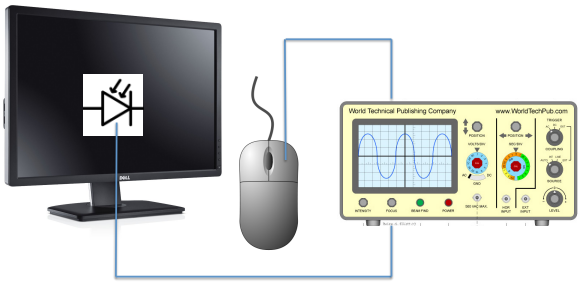


Figure 2: Sketch of demo setup.

3. DEMO SETUP

Measuring delays between input and output accurately requires a purpose-built setup. Previous, informal work as described in section 5 measured delay by filming the screen and a button with a high-speed camera while pushing the button repeatedly. This approach has some limitations. First, it is limited to the capture rate of the camera. Secondly, it is difficult to judge exactly when the button was pushed, and lastly it requires tedious manual work to analyse the videos.

3.1 Exact measurements

To get more exact results this demonstration will use an oscilloscope to measure the timing difference, see figure 2. The left most button is connected directly to the oscilloscope using an additional wire from the switch. A photosensor is held on the screen, and gives a signal when something bright appear. These signals are fed to the two channels of the oscilloscope. The participants find a dark place in the game. Then can they trigger an in-game effect that lights up the screen fast. Firing a weapon will in most games achieve this. Participants can thus see the delay between the two evens with high accuracy, by comparing the flank representing the mouse click (yellow on figure 3) with the flank representing change in light from the screen (blue on figure 3).

Our setup is independent of the hardware and software used to play the game. This allows testing real, commercial games, without modifying software or hardware. Results will represent the sum total of all delays. Delay from the input device, the game software, graphics card drivers, the graphics card itself or the screen are all added up. Investigating how much delay each part of the test system introduce requires modifying the test system itself locking the demo setup to one pre-modified testbed rig. We want participants to be able to test the delay of computers they bring to the setup to document the differences shown by a wide range of setups.

3.2 Cloud games

To test cloud games, we will present exactly the same setup, but participants play the game remotely through a commercial cloud gaming service. In parallel we will also show the network latency to the cloud gaming provider in realtime, so participants can see how large proportion of the delay is due to network latency. This part is contingent on access to cloud gaming services at the venue of the demonstration.

Figure 3: Capture of trigger event. Yellow line represent output from the mouse, blue line represents output from the photosensor. Both are active low. The difference in time between the two flanks are shown in green in the bottom-left corner of the oscilloscope screen.

4. PRELIMINARY RESULTS

To evaluate how well this setup works, and highlight how the results can be useful, we ran some preliminary experiments. For each condition we repeated the test 10 times, reporting minimum, maximum and average results. We used only one system for the purposes of this demo, planning a more exhaustive analysis of a wide range of systems in future work. This system was a MacBook Pro (Retina, 15-inch, Early 2013)[1], running OSX for the rectangle demo and Windows 7 for the game.

First, we ran a simple program that renders a rectangle in OpenGL in response to mouse-clicks. We compare this program running with and without vertical synchronisation. With vertical synchronisation it ran at about 60 FPS, without vsync the framerate fluctuated between 400 and 700 FPS.

Next we ran the game *Unreal Tournament 3* (UT3) and timed a basic pistol shot, an event that is supposed to be *immediate*. For this game we tested using three different conditions. Firstly, we tested using default settings. These had vsync off and a resolution of 1024 by 768. At these settings the game ran at about 60 FPS. Then we simply turned on vertical synchronisation and ran the experiment again, now the framerate was stable at exactly 60 FPS. Lastly we ran without vsync and with all settings optimised for faster framerate. These kind of tweaks are popular with professional gamers. As we see in table 2, response time is highly variable, even in the same game running on the same setup. Average delays in UT3 vary from 33 ms to 95 ms depending on the settings.

5. RELATED WORK

We have not been able to find any references to scientific publications measuring the local system latency for games.

[1]2.4 GHz Intel Core i7, 16 GB 1600 MHz DDR3, NVIDIA GeForce GT 650M

Table 2: Results from experiments.

Experiment	Avg.	Min.	Max.
Rectangle, vsync on	82 ms	73 ms	102 ms
Rectangle, vsync off	27 ms	21 ms	41 ms
UT3, default, vsync off	58 ms	52 ms	66 ms
UT3, default, vsync on	95 ms	79 ms	102 ms
UT3, optimised, vsync off	33 ms	23 ms	38 ms

The community for hardware component benchmarking and game optimisation, however, have touched on the topic on occasion.

Rendering Pipeline [15] finds values of between 76 ms and 112 ms for the game *Half Life 2* using realistic settings. *Blur Busters*, test multiple games [7]. They get results of 72–77 ms for *Battlefield 4*, 52–59 ms for *Crysis 3* and 22 ms to 40 ms for *Counter Strike: Global Offense*.

These numbers indicate that different games handle input in very different ways. Also, we know that network delays as low as 50 ms affects player performance in some games [9]. It is therefore important to investigate how player performance is affected by local delays of similar magnitudes.

6. CONCLUSIONS

When evaluating the effect of latency on the perceived performance of game systems, it is important to consider that the network delay is added to the delay of the local system. Preliminary results from this demo show delays of up to 100 ms. This shows that not only is the local delay at the same order of magnitude as network delays, it might in many cases be *larger*.

When discussing the tolerance levels for latency in networked games, both cloud gaming and traditional games, care should be taken to know the local delay as it may vary greatly between platforms depending on both hardware and software, in addition to software configuration.

This local latency may influence the measured tolerance levels in QoE experiments and bias the numbers reported as tolerance levels. It is therefore important that studies of the effects of other types of delay measure the inherent system delay when presenting numbers on how delays affect players. We propose using the setup presented in this demo to calibrate such experiments. Without such measurements it is very difficult to compare results from different studies on response time requirements.

In the future, we would like to investigate an array of different games and hardware under varied conditions. This will increase our understanding of delays in local gaming systems, and could likely be important in related fields such as Human-Computer Interaction (HCI).

Acknowledgements

The work in this paper was part-funded by the European Community under its Seventh Framework Programme through the Reducing Internet Transport Latency (RITE) project (ICT-317700) and part funded by the Research Council of Norway under the "TimeIn" project (No: 213265). The views expressed are solely those of the author(s).

7. REFERENCES

[1] Amin, R., Jackson, F., Gilbert, J., Martin, J., and Shaw, T. Assessing the Impact of Latency and Jitter on the Perceived Quality of Call of Duty Modern Warfare 2. In *Human-Computer Interaction. Users and Contexts of Use*, M. Kurosu, Ed., vol. 8006 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 97–106.

[2] Apple. Apple Thunderbolt Display. `https://www.apple.com/displays/specs.html`.

[3] Armitage, G. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *The 11th IEEE International Conference on Networks 2003 ICON2003* (2003), IEEE, pp. 137–141.

[4] Asus. Asus ROG SWIFT PG278Q. `http://www.asus.com/Monitors/ROG_SWIFT_PG278Q/specifications/`.

[5] Beigbeder, T., Coughlan, R., Lusher, C., Plunkett, J., Agu, E., and Claypool, M. The effects of loss and latency on user performance in unreal tournament 2003. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games* (New York, NY, USA, 2004), NetGames '04, ACM, pp. 144–151.

[6] Bernier, Y. Latency compensating methods in client/server in-game protocol design and optimization. *Game Developers Conference 98033*, 425 (2001).

[7] BlurBusters. Preview of NVIDIA G-SYNC. `http://www.blurbusters.com/gsync/preview2/`.

[8] Claypool, M. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks 49*, 1 (Sept. 2005), 52–70.

[9] Claypool, M., and Claypool, K. Latency and player interaction in online games. 40–45.

[10] Claypool, M., and Claypool, K. Latency Can Kill : Precision and Deadline in Online Games. *Proceedings of the First ACM Multimedia Systems Conference* (2010).

[11] Claypool, M., and Finkel, D. The Effects of Latency on Player Performance in Cloud-based Games. In *NetGames 2014(in print)* (2014).

[12] Dell. Dell UltraSharp 24 Ultra HD Monitor. `http://accessories.us.dell.com/sna/productdetail.aspx?sku=860-BBCD`.

[13] Jarschel, M., Schlosser, D., Scheuring, S., and Hoss feld, T. An Evaluation of QoE in Cloud Gaming Based on Subjective Tests. *2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (June 2011), 330–335.

[14] Li, S., Chen, C., and Li, L. Evaluating the Latency of Clients by Player Behaviors in Client-Server Based Network Games. *2008 3rd International Conference on Innovative Computing Information and Control* (2008), 375–375.

[15] Measuring Input Latency. Measuring Input Latency. `http://renderingpipeline.com/2013/09/measuring-input-latency/`.

Paper III

Measuring Latency in Virtual Reality Systems

Measuring Latency in Virtual Reality Systems

Kjetil Raaen[1 2 3], Ivar Kjellmo[1]

[1]Westerdals - Oslo School of Arts, Communication and Technology, Oslo, Norway
[2]Simula Research Laboratory, Bærum, Norway
[3]University of Oslo, Oslo, Norway

Keywords: Latency, Virtual Reality, Framerate, Mixed Reality

Abstract. Virtual Reality(*VR*) systems have the potential to revolutionise how we interact with computers. However motion sickness and discomfort are currently severely impeding the adoption. Traditionally the focus of optimising VR systems have been on frame-rate. Delay and frame-rate are however not equivalent. Latency may occur in several steps in image processing, and a frame-rate measure only picks up some of them. We have made an experimental setup to physically measure the actual delay from the user moves the head until the screen of the VR device is updated. Our results show that while dedicated VR-equipment had very low delay, smartphones are in general not ready for VR-applications.

1 Introduction

The last years have seen a great increase in interest and popularity of Virtual Reality (VR) solutions. Some are dedicated hardware made specifically for VR, such as Oculus Rift. Others are Mobile VR solutions such as Samsung's Gear VR, HTC and Valves' new Steam VR and even the simplified solution Google Cardboard. Providing a stereoscopic view that moves with their head, these systems give users an unprecedented visual immersion in the computer generated world.

However, these systems all have in common the same potential problems when it comes to motion sickness and discomfort when using the VR solutions[2]. An important source of these problems is delay. This paper presents an apparatus for measuring delay as well as detailed measurements of delay in some popular VR systems.

Frame-rate is a measurement of how fast frames are sent through the rendering pipeline. Delay on the other hand defines the time it takes from a user triggers an action, such as turning the head, until results are visible on screen. Previous work by one of us[7] has, however indicate that there is no linear relationship between frame-rate and delay in traditional computer graphics setups. From this, we assume that frame-rate is not the best metric. If VR solutions work like other graphics output devices, there is a significant difference between frame-rate measured inside an application and actual update speed.

By using a light sensor and an oscilloscope we hope to measure the exact delays in VR applications. Finding out how much delay there is in virtual reality headsets would be helpful in when investigating the causes for motion sickness in virtual worlds, as well as providing guidance to people producing software for these systems.

2 Background

Previous research has suggested various methods for measuring delay in VR systems all the way back to the previous VR-hype in the early nineties. Earlier work has used cameras [6] or light sensors [8]. What they all have in common is that they rely on a continuous, smooth movement of the tested devices. DiLuca et al.[3] summarise a series of them, and suggest their own approach. Their approach is the most elegant we have found so far.

2.1 Sources of delay

This section discusses the parts of the VR-pipeline that add most to the response delay. Components in the pipeline from the time the user moves until displays update are in general *black boxes*; documentation about how they work is often lacking. Thus, the only way to evaluate these delays is by measuring.

Screens used to display output add some delay, which can be divided into two parts: *screen refresh* and *response time*. LCD screens used in VR displays receive and display updated images at a fixed rate. This rate is termed *screen refresh* rate. Most modern screens update at 60 frames per second (FPS), or every 16.7 ms. *Response time* denotes the time the physical pixels take to change colour.

A *frame buffer* is memory used for holding a rendered frame about to be displayed on the monitor or VR-display. Modern renderers use at least two frame buffers. To avoid showing unfinished frames to the user, drawing happens in one buffer, while the other is being displayed. This practice is termed *double buffering*. Further, to avoid showing parts of two different buffers, it is common to wait for the next screen refresh before swapping. When double buffering is used, rendering follows the sequence: Assume frame 0 is the frame during which an event from an input device is registered. Frame 1 contains the result of the event, and at the time of frame 2 the result is sent to screen. This gives a minimum of 1 full frame time from input event to result on screen.

At 60 FPS this adds up to a minimum of one frame delay (17 ms) to a maximum of two frames (33 ms) delay. Further, not all hardware is capable of keeping up with the target frame-rate at all times. Slow hardware leads to significantly longer delays. An increased number of frame buffers in the pipeline increases this delay, because more steps are added between rendering and displaying data.

Dedicated VR equipment has tailor-made hardware and drivers that have been tweaked for low latency performance. The gyros of smartphones on the other hand are mainly intended for navigation or keeping the screen rotated the correct way, neither of which require fast response time or high accuracy.

2.2 Acceptable delay

Empirical values for how long delay is acceptable is difficult to find. Davis et al. investigated a condition they term *cybersickness*[2] in analogy to motion sickness. They consider a range of options for the cause of these problems, delay among them. However they do not quantify delay that might lead to symptoms. Jarods [5] concludes that people reacts very differently to latency. Some people would hardly notice a 100 ms delay while other very sensitive people are able to perceive down to 3-4ms of latency.

Most design guidelines and measurements reports are from developers and of VR equipment and VR-software rather than scientists. John Carmack [1] states that a latency of 50 ms feels responsive but the lag when moving in the virtual world is noticeable. He recommends that latency should be under 20 ms. Neither methodology or background numbers and test results are presented.

3 Experiment Setup

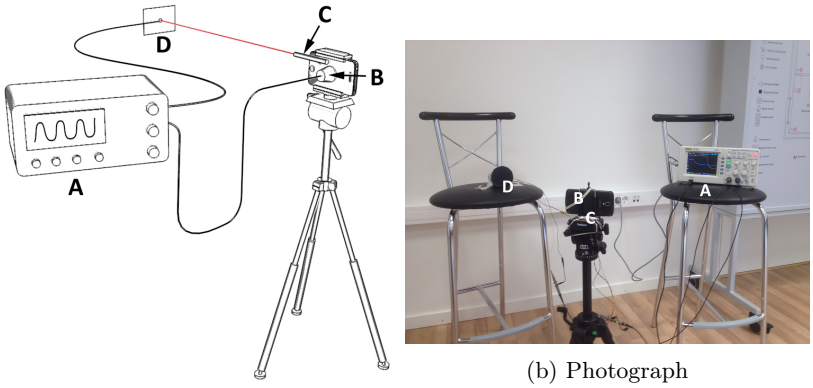
To measure response time after abrupt movements, we need the virtual reality device to run a program that detects a small rotation and as fast as possible change the displayed picture. A simple program that detects rotation and changes the displayed picture would create solve this problem. However, we are interested in delays from actual 3D virtual reality software. Therefore we used a popular game engine, Unity 3D[1], and made a scene that creates abrupt changes based on headset movement. We tested multiple VR devices. Oculus Rift is a dedicated VR display solution, designed for this purpose. The other systems are smartphones, which developers have discovered have all the required hardware to run VR application and can also function as headsets.

The physical setup(fig 1) consists of the VR device (Oculus Rift, Smartphone etc.) mounted on a camera tripod. One light sensor is attached to the screen of the VR-device to register the virtual scene shifting from white to black. A laser pen is also attached to the virtual device pointing at another light sensor approx. one meter from the tripod setup. Both light sensors are connected to an oscilloscope. When we move the VR device by turning the tripod, the light sensor illuminated by the laser pen registers the disappearance of the light. When the movement is detected, light sensor connected to the virtual device screen measures the light shift from the white plane disappearing in the virtual scene.

This allows us to measure the time from when the physical movement starts until in movement is visible in the virtual scene. We measured 5-10 times on each device and we present an average of the measured values.

The virtual setup running on the VR-device is a simple completely black virtual 3D scene with a white self-illuminated plane and a virtual stereo camera. The virtual camera is set up with normal VR movement controllers. For the mobile phone setup the Durovis Dive SDK[2] was used while for the Oculus rift setup

[1] http://unity3d.com
[2] http://www.durovis.com/sdk.html



(a) Schematic drawing

(b) Photograph

Fig. 1: The Physical setup with the Oscilloscope (A), light sensor (B) attached to Virtual device, Laser pen (C) and light sensor (D) picking up the laser.

the Oculus Rift SDK[3] was used. This in order to set up a virtual reality scene with an absolute minimum of content optimised for a highest possible frame-rate, giving consistent values of multiple hundred frames per second and to use the build in VR movement controllers. The OculusRift devices were connected to a fast laptop[4].

The virtual scene consists of a white plane on a black background. The camera faces the white plane from a large distance and is set to a very narrow field of view. The scene is tweaked so that the display is completely white initially. Because of narrow field of view the camera is very sensitive to rotation. This means that even a small rotation in the headset leads to the screen changing colour from white to black, a change picked up by the light sensor.

4 Results and discussion

Measured delay is simply the time from the light sensor illuminated by the laser changes from bright to dark until the light sensor attached to the screen reports the same change. Table 1 shows the results from the systems we tested. Unsurprisingly, the dedicated hardware has much faster response rate. Further, the smartphones do not give developers access to control the vertical synchronisation setting, and it seems from the numbers that it is always on.

For the dedicated VR displays, v-synch has a large effect on total delay. While the Virtual scenes with V-sync off the frame-rate in the application was as high as 3000 fps, the one with V-sync on would hold a steady frame-rate at 60 fps.

[3] https://developer.oculus.com/

[4] Windows 7, Intel i7 - 3740 QM CPU @ 2,70 ghz, NVIDIA Quadro K2000M - 2048 mb DDR3

Turning off this feature introduces introduce visual artefacts, but reduces the delay significantly. Without vertical synchronisation, these products can react very fast. Smartphones on the other hand are much slower, with delays close to 100 ms for most models. The exception is the Samsung S5 which has a delay less than 50ms. This result is similar to the Oculus Dev kit 2 with V-sync on. The screen in the Samsung S5 is actually the same as in the Oculus Dev kit 2 and the result confirms the similarity between the two devices.

These results come with some caveats. Despite our efforts, the rendered scene might not turn instantly from white to black between one frame and the next. Intermediate frames should show up as plateaus in the oscilloscope output, but we cannot be completely sure the animated movement did not show a frame or a few of parts of the white square. However, consistent results indicate that this is unlikely.

Table 1: Results from delay measurements.

VR Display	Avg.	Min.	Max.
Oculus Rift dev kit 1, v-sync ON	63 ms	58 ms	70 ms
Oculus Rift dev kit 1, v-sync OFF	14 ms	2 ms	22 ms
Oculus Rift dev kit 2, v-sync ON	41 ms	35 ms	45 ms
Oculus Rift dev kit 2, v-sync OFF	4 ms	2 ms	5 ms
Samsung Galaxy S4(GT-I9505)	96 ms	75 ms	111 ms
Samsung Galaxy S5	46 ms	37 ms	54 ms
iPhone 5s	78 ms	59 ms	96 ms
iPhone 6	78 ms	65 ms	91 ms

5 Conclusions and future work

We have presented a simple and precise solution for measuring delay in VR-systems. In contrast to DiLuca's [3] and other previous work, our system are able to detect delays in instantaneous, jerky movements. Earlier systems have generally relied on smooth, continuous movements, which are quite alien to a real user in a chaotic game. Jerald finds in 2012 [4] that users are most sensitive to delays when their movement changes direction. Current and future VR technology often employ prediction to reduce delay during continuous motion, which does work when motion changes. Thus, measuring delay for these sudden movements produces results more aligned with the sensitivity of users. Our setup does not require any modifications to the VR-systems to measure their delay, and can thus easily be applied to new hardware as soon as it becomes available.

Regarding how short delay is *good enough* both developer guidelines and than scientific papers mostly agree, placing ideal delay at less than 20 ms[4]. Even if the numbers comes from a simulated environment it gives a quality study of human perception when it comes to noticing latency.

With this in mind, it seems clear that the phones are too slow. Both of the latest models of iPhone, as well as the older Samsung have latencies around 100 ms,

which all sources agree is far too slow. The older phones are not designed to run VR systems, but these results give us a picture of the difference of yesterday's standards and the present specifications needed for pleasant VR experiences on mobil. In this picture the Samsung Galaxy S5 is closer to acceptable. Depending on which limits you use, an average value of 46 ms is either barely acceptable or somewhat too slow. However, Samsung markets some phones clearly as VR-devices, such as Samsung Note 4 and the upcoming Samsung Note 5. These are currently the only devices working with Gear VR, their new system for combining phones with VR. Our results from the Galaxy S5 indicates that some optimisations for VR are already included in this phone.

Oculus Rift in both versions respond extremely fast with v-synch off, fast enough to satisfy most guidelines. With synchronisation on, on the other hand, they are barely fast enough. The same guidelines that recommend 20 ms delay also claim v-synch is important for user experience. We found no way of satisfying both the delay requirement and the requirement to use vertical synchronisation at once. Oculus Rift is advertised to contain a built in latency tester. We did not find documentation on how to use the, nor did we find any description on how it works. Comparing our results with the output of this hardware would be interesting.

Development of new VR technologies, both dedicated and mobile, is progressing at a rapid pace these days, and not all the systems presented here will be current by the time this paper is published. The experiment setup on the other hand should be useable for any new system, and we hope to update our data with new systems when they become available. Other factors influencing user experience should also be studied in more detail, as well as their interaction with delay.

References

1. Carmack, J. Latency Mitigation Strategies. https://www.twentymilliseconds.com/post/latency-mitigation-strategies/, 2013.
2. Davis, S., Nesbitt, K., and Nalivaiko, E. A Systematic Review of Cybersickness. In *IE2014* (New York, NY, USA, 2014), IE2014, ACM.
3. Di Luca, M. New Method to Measure End-to-End Delay of Virtual Reality. *Presence: Teleoperators and Virtual Environments 19*, 6 (2010), 569–584.
4. Jerald, J., Whitton, M., and Brooks, F. P. Scene-Motion Thresholds During Head Yaw for Immersive Virtual Environments.
5. Jerald, J. J. *Scene-Motion- and Latency-Perception Thresholds for Head-Mounted Displays*. PhD thesis, 2009.
6. Kijima, R., and Ojika, T. Reflex HMD to compensate lag and correction of derivative deformation. *Proceedings IEEE Virtual Reality 2002 2002* (2002).
7. Raaen, K., and Petlund, A. How Much Delay Is There Really in Current Games ? *ACM MMsys* (2015), 2–5.
8. Swindells, C., Dill, J., and Booth, K. System lag tests for augmented and virtual environments. *Proceedings of the 13th annual ACM . . . 2* (2000), 161–170.

Paper IV

Can gamers detect cloud delay?

Can gamers detect cloud delay?

Kjetil Raaen
Westerdals — Oslo School of Arts, Communication and Technology
Faculty of Technology
Simula/University of Oslo
Email: raakje@westerdals.no

Ragnhild Eg
Simula Research Laboratory
1325 Lysaker, Norway
Email: rage@simula.no

Carsten Griwodz
Simula Research Laboratory
1325 Lysaker, Norway
University of Oslo
Email: griff@simula.no

***Abstract*—In many games, a win or a loss is not only contingent on the speedy reaction of the players, but also on how fast the game can react to them. From our ongoing project, we aim to establish perceptual thresholds for visual delays that follow user actions. In this first user study, we eliminated the complexities of a real game and asked participants to adjust the delay between the push of a button and a simple visual presentation. At the most sensitive, our findings reveal that some perceive delays below 40 ms. However, the median threshold suggests that motor-visual delays are more likely than not to go undetected below 51-90 ms. These results will in future investigations be compared to thresholds for more complex visual stimuli, and to thresholds established from different experimental approaches.**

I. Introduction

Games resemble real life in many ways, and players expect them to behave like the physical world. There, most actions we perform lead to instant reactions. Unfortunately, because of technical restrictions, immediate results are impossible in computer games. To study how we perceive these delays between input actions and visual results, we apply work on sensory interactions to current gaming shortcomings imposed by network limitations.

Our sensory systems process many external stimuli at the same time, but somewhere along the way they converge and align to create a unified experience. For instance, in many human-computer interactions, a button push and a visual event have to coincide in order to ensure fluent operations. Fortunately, they need not be in perfect synchrony. Humans learn from experience what to expect following a familiar action, moreover, the perceptual system can compensate for, and even adapt to, short time displacements [1]. Causality is an important factors in motor-visual interaction, if too much time passes after an action, the delayed consequence may be attributed to another event [2]. Online games introduce concerns related to causality and anticipation, considering that gamers expect immediate reactions from their actions. Unfortunately, network limitations slow down the reaction time, creating temporal delays between the motor and the visual signals. Eventually, the delays become too long for the perceptual system to compensate, and they become detectable.

Multiplayer games communicating over a network exhibit two types of delay. *Interface delay* is the most critical, but shortest, occurring between a user's input and the resulting visual presentation. Further, when games are played across a network, information exchange between client and server, in addition to processing on the server, introduces delays. This type of *network delay* can extend to tens or even hundreds of milliseconds, and developers strive to hide it using various techniques [3]. Recently, a new way of delivering games has gained popularity: *Cloud gaming*, the concept of running the entire game remotely and using the local computer as a dumb terminal. In this scenario, network delays appear between input and output, and add to the interface delay. Latency hiding techniques have become much more difficult to implement, which highlights the importance of understanding users' basic latency tolerance and ability to detect it. Jarschel and colleagues [4] ran a study on player sensitivity to latency in cloud gaming, but their manipulations included no latencies shorter than 80 ms. Because subjective quality of experience was noticeably reduced at this value, their results emphasise mainly how sensitive players are to fairly short delays.

Psychologists typically tackle their research questions using controlled experimental designs that heed statistical power. On the other hand, game developers tend to use rules-of-thumb and estimates based on experience when working to reduce lag in networks and computational processes. In the study of real game scenarios, stringent experimental methods with repeated presentations could require participants to spend hours on a single experiment. To circumvent this, earlier studies on in-game delays have applied the experimental method to real games, and have instead restricted the number of participants [5]. With a smaller pool of participants to average across, generalisations come with a note of caution. Our motivation is to empirically establish thresholds for detectable motor-visual delays, and in this first step we explore their temporal interaction in isolation.

II. Experiment design and method

We designed and conducted a behavioural experiment that uses a button device and a simple visual stimulus, and we ran it on standard computers[1]. The button devices are Griffin click+spin USB controllers, or jog shuttles, which are designed solely for button-pushes and left/right rotations[2]. The visual stimuli consisted of a black disc on a white background that would flash on or off in response to the button pushes. We wanted to explore object size as an additional factor that could shape the perceptual process. Thus we included both a small and a large disc, with diameters of 20 and 200 pixels.

[1]HP Z200 (Intel Xeon X3430 CPU 4 cores@2533MHz, 8GB RAM) running Windows 7 and connected to Acer AL1916W monitors with1440x900 screen resolution and 60 Hz refresh rate

[2]http://store.griffintechnology.com/powermate

978-1-4799-6882-4/14/$31.00 ©2014 IEEE

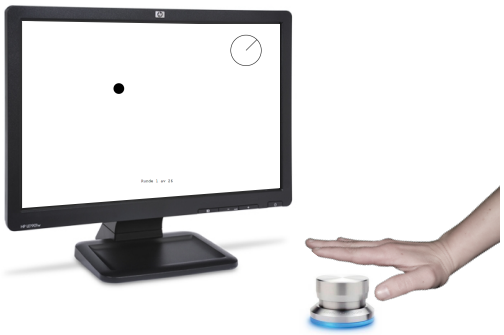


Fig. 1. Illustration of the experimental set-up with the USB controller placed conveniently in front of the participant and the visual stimulus presented on the monitor. The visual stimulus is here represented by a 20 pixel black disc.

At a viewing distance of 60 cm, these disc sizes correspond to visual angles of 5.6° and 0.56°, respectively. During the experiment, a visual guideline in the form of a simplified clock face helped participants keep track of rotations made with the USB controller. An example of the experimental set-up is visualised in Figure 1. The initial delay at which stimuli were introduced varied between 200 ms, 300 ms, and 400 ms. Each stimulus condition was repeated four times, so that participants completed a total number of 24 trials, in addition to two initial practice trials. The experiment took approximately 10-15 minutes to complete, including an initial questionnaire that assessed possible background variables, such as gaming experience.

We recruited 13 female and 28 male volunteers, aged between 19 and 43 years (mean = 24), and conducted the experiment in a computer lab at the Norwegian School of IT. Participants were instructed to click the knob of the USB controller in order to make the disc presented on the monitor flash on or off, and to rotate the controller when they wanted to adjust the temporal delay between the push and the disc-flash. The delay would change proportionally to the adjustment angle. Because the controller provided no reference points, participants were always unaware of the value of the delay and the direction of their adjustments. Values for delay could be adjusted from 0 to 500 ms; if adjusted past these extremes, the values would gradually decrease or increase away from the extreme. We emphasised that they could spend as much time and make as many pushes and adjustments as they wanted. When they were satisfied that they could no longer perceive the temporal delay, they proceeded to the next trial by pressing the spacebar.

An experimental set-up that depends on a computer system is bound to be influenced by physical and computational limitations. We highlight these limitations in order to explain the precautions taken. First of all, the screen refresh rate of 60 Hz could introduce up to 16.7 ms visual lag. Assuming a random distribution of clicks between screen updates, this corresponds to an average of 8.3 ms delay. Clicks collected by the USB controllers also have limited temporal resolutions, the experiment machines polled these at a rate of 125 Hz. This equals a maximum delay of 8 ms and an average of 4 ms. In total, these uncertainties add up to 12.3 ms average and 24.7 ms maximum technical delay. An additional uncertainty relates to the experimental task. A standard approach would be to adjust delays continuously downwards, meaning that the exact detection threshold could be surpassed. Accepted values may therefore correspond to a point below the threshold. In the worst case, our collected data may be uniformly distributed over a region of imperceptible delays. Finally, the controller rotations adjusted the delays at 25 ms increments, which leads to a clustering of the collected data. These concerns are all addressed in the interpretation and presentation of our findings.

III. Results

In our study of the perception of motor-visual delays, we designed and conducted an experiment where participants adjusted the delay between a button-push and a visual disc flash. We also explored the relationship between accepted delay scores and the size of the presented disc, as well as participants' gaming experience.

Any initial delay value that participants accepted without adjustments was categorised as an accidental accept; accidental accepts were thus labelled as missing values and treated like such for the main analyses. We also judged that participants who made two or more of these accidents (corresponding to approximately 5% of all trials) did not adhere to the experimental procedures and we therefore excluded the data from two participants from the analyses.

We explored potential effects with Wilcoxon signed rank-sum tests, but found no significant differences in accepted delays due to disc size ($W = 660, n = 38, Z = 0.06, p > .5$) or to gaming experience ($W = 89189, n = 893, Z = -0.46, p => .3$). However, we observed great differences in task effort between participants, defined by the number times they pushed the button. Using the number of button-pushes as an estimate for the number of adjustments made, we ran a linear regression with a logarithmic fit. This revealed a significant, but negative, relationship between task effort and accepted delays ($accepted\ delay = -19.96 * log(clicks) + 118.14, R^2 = .04, p < .001$). In other words, frequent adjustments led to lower delay values. Furthermore, we found that experienced gamers made significantly more adjustments than those with less experience ($W = 66254, n = 893, Z = -5.31, p < .001$), meaning that experienced gamers make more attempts than less experienced gamers before finding an acceptable motor-visual delay. Accordingly, we surmised that the accepted delays depended in part on the number of adjustments made, contributing to greater temporal sensitivity among those most dedicated to the task. In light of this finding, we decided to run separate analyses for the full participant group and for the best-effort subgroup. The latter group was defined by their average number of adjustments, and it includes all participants who made more than the median number of 18 button-pushes during a trial.

To establish detection thresholds for motoric-visual temporal delays, we plotted an empirical cumulative density distribution with all delay scores and then established the best-fitting gamma distribution. From this distribution, we derived the 25th, 50th, and 75th percentiles. We did the same for the best-effort sub group. The distributions and percentiles are portrayed in Figure 2. When interpreting these results, keep in mind the limitations outlined in section II. Notably, even when taking these cautionary measures, the 25th and 50th thresholds still fall below 100 ms.

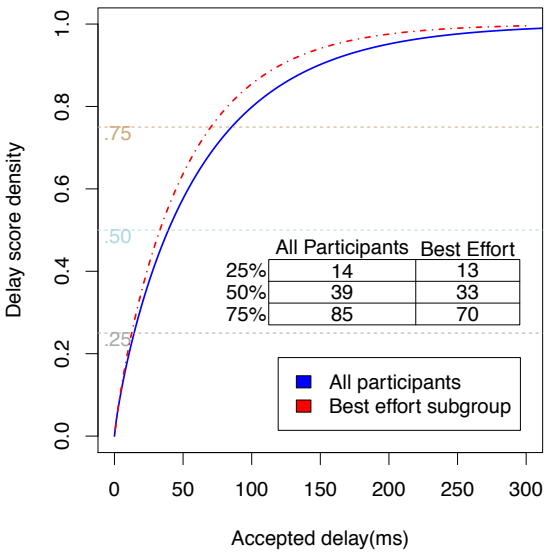


Fig. 2. Cumulative density of participants' accepted delay scores plotted with the best-fitting gamma distributions. The thresholds listed in the table correspond to the 25th, 50th and 75th percentiles of the distributions.

IV. Discussion

This work presents the first step in our studies of the interplay between human actions and digital events, where we aim to establish thresholds for detectable motor-visual delays. The applicability of these perceptual thresholds is particularly prominent in current challenges facing the gaming industry.

In our first venture, we explored the detectability of these delays under the most ideal conditions, using simple, isolated stimuli. We also considered individual differences, although we found no correlation between gaming experience and accepted delays. Similarly, we found no effect related to the size of the visual object. Instead, our results showed that some participants made more adjustments during a trial than others, and this effort was reflected in decreased values for accepted delays. Moreover, experienced gamers had a greater tendency to make more adjustments than non-experienced gamers.

Because the experiment task allowed participants to adjust delays below the actual point of detection, we prefer to err on the side of caution. Our cautionary measures take into account that the collected data could be uniformly distributed across the individuals' detection ranges, as well as the average 12 ms delay added by our system.

When incorporating these limitations into our motor-visual delay thresholds, we find that a small share of participants cannot perceive visual lags shorter than 97-182 ms, while those whose scores fall below the median are able to detect delays around 51-90 ms. The most sensitive of our participants could even perceive visual lags as short as 26-40 ms. Because of the strong correlation between task-effort and accepted delay, we included a separate analysis for participants who made more adjustments than the median of 18. For this half of participants, the median accepted delay lies between 45-78 ms. Although these thresholds allow room for uncertainty, they serve as guidelines to the sensitivity of the human perceptual system when encountering motor-visual delays. Importantly, they suggest that a large proportion of our participants can easily perceive delays shorter than 100 ms. Moreover, for one out of every four trials, our participants could even detect delays below 40 ms.

In cloud gaming scenarios, all network latency appears as interface latency. The presented experiment tackles this temporal challenge using a button-device and a simple visual presentation, and herein lies the most pronounced difference between real-life cloud games and our experimental scenario. Most games involve reactions to moving stimuli, and this points the direction for our next step in the study of motor-visual delay perception. For the time being, the isolation of the motor-visual interaction allowed us to investigate how much delay people can detect when they are at the most sensitive. By presenting game developers with a lower bound for acceptable motor-visual delays, we hope to introduce a level of confidence in the development stage. Below the shortest of our established thresholds, gamers are very unlikely to consciously experience any visual lags. On the other hand, gaming performance may still be negatively affected by imperceptible delays, which is another issue we aim to address in future works.

V. Conclusions and Future Directions

If our participants are representative of a larger population of regular computer users, we can assume that half of these individuals will have a difficult time tolerating services that operate with up to 100 ms delay. At the most sensitive, some of these people are also able to detect motor-visual delays as short as 26-40 ms. Consequently, providers of cloud gaming services should bear in mind that some of their players could be very sensitive to the visual consequences of network latency.

With this foundation, we now have a scale of noticeable delays to build our work on. In the next planned step, we will compare experimental methodologies to ensure that the established thresholds do not merely reflect the assigned task. We also plan to apply more dynamic and more complex stimuli, to explore whether this could alter the detection of motor-visual delays. Our end objective is to conclude the project with a fully operational game; this will build the foundation for an investigation into the ability of game players' to compensate for lags.

Acknowledgements

We acknowledge the valuable feedback provided by Professors Mark Claypool, Pål Halvorsen, and Dawn Behne.

References

[1] C. Stetson, X. Cui, P. R. Montague, and D. M. Eagleman, "Motor-sensory recalibration leads to an illusory reversal of action and sensation." *Neuron*, vol. 51, no. 5, pp. 651–659, Sep. 2006.

[2] J. Heron, J. V. M. Hanson, and D. Whitaker, "Effect before cause: Supramodal recalibration of sensorimotor timing," *PloS One*, vol. 4, no. 11, p. e7681, Jan. 2009.

[3] Y. Bernier, "Latency compensating methods in client/server in-game protocol design and optimization," *Game Developers Conference*, vol. 98033, no. 425, Mar. 2001.

[4] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, "An Evaluation of QoE in Cloud Gaming Based on Subjective Tests," *Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 330–335, Jun. 2011.

[5] M. Claypool and K. Claypool, "Latency and Player Actions in Online Games," *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, Nov. 2005.

Paper V

Instantaneous human-computer interactions: Button causes and screen effects

Instantaneous human-computer interactions: Button causes and screen effects

Kjetil Raaen[123] and Ragnhild Eg[1]

Westerdals - Oslo School of Arts, Communication and Technology, Oslo, Norway
Simula Research Laboratory, Bærum, Norway
University of Oslo, Oslo, Norway

Abstract. Many human-computer interactions are highly time-dependent, which means that an effect should follow a cause without delay. In this work, we explore how much time can pass between a cause and its effect without jeopardising the subjective perception of instantaneity. We ran two experiments that involve the same simple interaction: A click of a button causes a spinning disc to change its direction of rotation, following a variable delay. In our *adjustment experiment*, we asked participants to adjust the delay directly, but without numerical references, using repeated attempts to achieve a value as close to zero as possible. In the *discrimination task*, participants made judgements on whether the single rotation change happened immediately following the button-click, or after a delay. The derived thresholds revealed a marked difference between the two experimental approaches, participants could adjust delays down to a median of 40 ms, whereas the discrimination mid-point corresponded to 148 ms. This difference could possibly be an artefact of separate strategies adapted by participants for the two tasks. Alternatively, repeated presentations may make people more sensitive to delays, or provide them with additional information to base their judgements on. In either case, we have found that humans are capable of perceiving very short temporal delays, and these empirical results provide useful guidelines for future designs of time-critical interactions.

1 Introduction

Several of our work and after-work hours are spent typing and clicking with keys and buttons to tell a machine what to show on a screen. Sometimes, we don't pay much attention to the swiftness of the visual presentation; yet at times, we want the input to lead to instant results. However, due to screen refresh intervals, buffering and sometimes network stalls, our actions are not always immediately followed by the expected outcomes. The extent and acceptability of an outcome delay is highly context-dependent. In human-computer interactions,

the *responsiveness* of a system can vary from a few milliseconds to a few seconds. Seow separates between four categories of responsiveness [1], labelling the slowest interactions *flow* and *continuous*; both of these exceed one second. Somewhat faster are *immediate* responses, which range from half a second to one second, and the fastest are termed *instantaneous*. The latter category is recommended for graphical controls and other interactions that mimic the physical world, but the thresholds are only estimated to be between 100 and 200 ms. While delays shorter than 100 ms may be imperceptible, they can still affect user performance, for instance through increased stress levels [2]. This work sets out to establish empirical values for how fast an outcome must follow an input in order for a user to perceive it as *instantaneous*.

1.1 Interaction delays and sensory processes

Humans are very adept at handling and acting on objects, facilitated by both the motoric and the visual systems, along with other inputs. Sense of agency refers to the experience of being the direct cause of an event, and this term encompasses the expected delays that follow many actions [3]. Indeed, one study found that participants maintained the sense of agency from a joystick controlling the movements of an image for intervals as long as 700 ms [4]. This type of delay can approximate those found that follow real physical events, where consequences are stalled by the time taken to traverse a distance. However, many human-computer interactions involve series of inputs and outputs and these require far more speedy reactions. Whether typing in text, shooting at moving targets, or moving a cursor across the screen, most users expect instantaneous responses from the system. The higher demands for this type of human-computer interaction is emphasised by the findings of an experiment that compare the temporal boundaries for the sense of agency and the sense of simultaneity [5]. Participants were asked to push a button and watch for a visual flash, then make a judgement on the simultaneity of the events, or on the event serving as the agent. On average, the button-push was perceived as the agent as long as the visual flash did not lag by more than ≈400 ms; conversely, the two events were judged as simultaneous, at greater than chance rates, when the flash delay stayed below ≈250 ms [5]. In fast-paced game scenarios, similar delays become noticeable to players around 100 ms, and these can be detrimental to the gaming experience [6,7]. Furthermore, mouse actions that require pointing and dragging have been found to be even more sensitive to temporal delays [8]. Still, humans are capable of adapting to fairly long temporal delays (235 ms) between movements of a mouse and movements on a screen, although this becomes increasingly difficult as the visual task speeds up [9].

Clearly, instantaneous and simultaneous are not synonymous with zero delay, a computational impossibility. Yet, these and similar human-computer interactions place strong demands for speedy responses on a system. Moreover, studies

on multisensory and sensorimotor processes have demonstrated that the human perceptual system is adaptable and quite capable of compensating for short temporal offsets between corresponding signals [5,10,11,12]. In our quest to find out exactly how much visual lag the perceptual system can compensate for following a motoric input, we have run a series of behavioural experiments on motor-visual delays. Our initial investigations involved direct delay adjustments using a jog-shuttle, with the corresponding visual event presented as disc that flashed on or off on a screen [13]. This approach allowed participants to repeatedly test and adjust the delay by turning the wheel and clicking the button of the jog-shuttle. Results from this experiment revealed that people vary greatly in their sensitivity to this type of delay, but the established median threshold was still far lower than expected at 39 ms. Adding system limitations to this value, our first investigation concluded that humans are on average capable of perceiving motor-visual delays as short as 51-90 ms.

1.2 Discrimination and adjustment of delays

This study continues our investigations into human sensitivity to motor-visual delays. It addresses the question on the appropriateness of our initial experimental approach and puts it back to back with a more traditional approach. Hence, we compare thresholds derived from two distinct methodologies, aiming for a more expansive range of data to generalise from. Because our first paradigm [13] allows participants some leeway to get results lower than they can actually percieve, we selected an isolated experimental task that relies on subjective discrimination and a binomial response selection. The simultaneity judgement task is a common methdology in multisensory research [5,10,12], and like the name implies, it involves a judgement call on the simultaneity of two signals. In our version of this task, participants are asked to discriminate between a motoric input and a visual output and make a judgement on whether the output followed immediately or whether it was delayed. Our comparison of delay thresholds established from two different methodologies thus forms the basis for an ongoing discussion around the use of less traditional experimental methods. Furthermore, we build on our earlier experiment where the motoric input, the button-click, resulted in the appearance or disappearance of a black disc [13]. Thus, we extend our work by adding dynamics to the previously static presentation. In the two current experiments, the visual presentation is made up of a black disc that rotates continuously at a steady pace. Moreover, bearing in mind that fast visual presentations can affect performance on these types of tasks [9], we include two speeds of rotation. By doing so, we explore whether the speed of motion can influence not only performance, but also the sensitivity to motor-visual delays.

Fig. 1: Illustration of the experimental set-up.

2 Method

We explored subjective sensitivity to temporal delays between motor inputs and visual outputs in two repeated-measures experiments. The first experiment applied the described adjustment task, extending on our earlier work by replacing the static visual stimulus with a rotating disc. The second experiment encompassed the same visual presentation, but introduced a variation of the more common simultaneity judgement task [5,12,10], hereafter referred to as the *discrimination task*. We ran the two experiments over one session in a computer lab at Westerdals, with the order of presentation counterbalanced across our 10 female and 41 male participants (aged between 19 and 33 years).

We aimed to keep conditions as comparable as possible across the experiments, allowing for a direct comparison between the two methodologies. We therefore used the same visual stimulus throughout, simply a black disc moving in a continuous circle. The experiments ran on MacBook Pro computers with 15.4" monitors and participants' adjustments and responses were registered using Griffin click+spin USB controllers[1]. These are simple controllers called *jog-shuttles*, comprised of a big click-button that also serves a rotating wheel. In both experiments, the button served as a trigger to change the direction of the disc's rotation, as portrayed in figure 1. The disc rotated either slowly or quickly (0.2 or 1 revolution/second), with the speed of rotation varying randomly from trial to trial; the initial direction of rotation was also randomised across trials.

[1] http:/store.griffintechnology.com/powermate

2.1 Adjustment experiment

For the adjustment experiment, each trial commenced with an initial delay (100, 200, 300, or 400 ms). Participants were instructed to push the button on the jog-shuttle to change the direction of rotation and turn the wheel to adjust the delay between their push and the visual change. Due to the lack of reference points, participants were always unaware of the physical value of the delay; however, a clock-face served as a visual cue for the full range of delays. The task involved repeated adjustments and tests of the motor-visual delay, using the jog-shuttle, until the visual presentation was perceived to follow the motoric input instantaneously. Participants were allowed to spend as long as they wanted on each trial, but they had to make a minimum of ten button-clicks before proceeding to the next trial. Each trial therefore involved a series of wheel rotations and button-clicks before reaching the point of no delay, in a sequence illustrated in figure 2. With four levels of initial delay, two levels of rotation speed and two levels of rotation direction, along with two repetitions of all conditions, the full adjustment experiment included 32 trials and took approximately 10 minutes to complete.

2.2 Discrimination experiment

In the discrimination experiment, each trial involved a single button-click with a corresponding change in the direction of rotation. The delay between the click and the directional change varied randomly between 11 pre-established values (0, 20, 40, 60, 80, 100, 140, 180, 220, 260, and 300 ms). We asked participants to click button and pay close attention to the visual change. They were thereafter prompted for a judgement on the simultaneity of the motor and the visual events. Participants provided their responses by turning the wheel left or right to choose either the ”immediate” or the ”delayed” response options. We included four repetitions of all delay, direction and speed conditions, making a total of 176 experimental trials. With the short trial presentations, the experiment duration was on average 12 minutes.

2.3 Limitations

Our methodologies carry with them a few limitations. Even without system or network lags, computers will always introduce some delay in any interaction. These values are largely disregarded in the literature because they cannot be controlled in an experimental set-up. Although a computer's internal system does not have the functionality to calculate the total duration between user input and screen output, we have applied an external set-up to measure this delay. Using

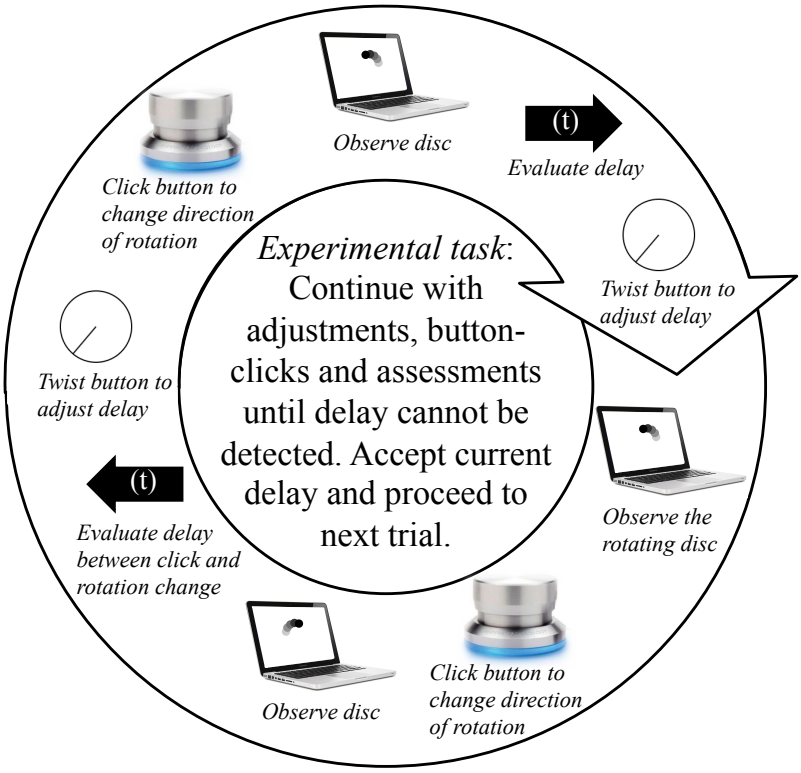


Fig. 2: Circular timeline illustrating the experimental procedure for the adjustment experiment. The experiment starts with an initial push of the button and a delayed change in rotation direction and it continues for as long as it takes the participant to adjust the delay down to an imperceptible level.

a light sensor on the screen and an extra connector to the button, we performed 10 measurements and obtained an average input-output delay of 51 ms, this procedure is described in more detail in [14]. We report all scores and thresholds without adding this delay, in order to allow for comparisons with earlier studies in the field. However, this number should be kept in mind for a better representation of the human ability to detect motor-visual delays. Moreover, adding this delay to our results also improves the ground for comparison with findings from studies that make use of purpose-built experiment hardware.

3 Results

We initially treated data from the two experiments separately. For every factor and delay level, we calculated each participant's mean and then derived the 50^{th} percentile threshold from their individual distributions. This statistical approach did result in thresholds that exceeded the presented delay values, but only for a few individuals who likely have high tolerance to these types of delay. Furthermore, we took the precaution of checking for outliers based on the discrimination task distributions. For the vast majority of participants, the rate of "immediate" responses decreased as delays increased. However, one participant's scores were discarded because the rate of "immediate" responses increased alongside the delay values, yielding a negative threshold value.

3.1 Adjustment task

We ran two Wilcoxon signed rank-sum tests to investigate potential variations in delay sensitivity between the two initial disc rotation directions and the two disc rotation speeds, we also ran a Friedman test to explore differences due to the initial delay values. None of the tests revealed significant differences between the conditions. Following this, we collapsed scores across presentation modes and established the overall median threshold for adjusted delays to 40 ms. The 25^{th}, 50^{th} and 75^{th} percentile thresholds are presented as an empirical cumulative distribution in figure 3. With our motivation to evaluate the appropriateness of two distinct experimental methodologies, we also established the mode value for delay adjustments from the density plot presented in figure 5a. From this distribution, we found that the mode falls around 30 ms, a lower value than the median threshold. Furthermore, we observed an asymmetrical distribution, where the majority of scores centered around the mode, but a long tail of delay scores extended close to 400 ms.

3.2 Discrimination task

For the discrimination scores, we derived a best-fit logistic regression model, see figure 4. Running another two Wilcoxon signed rank-sum tests, we found no significant differences between the rotation directions and the rotation speeds. Hence, we collapsed scores across presentation modes and established the discrimination threshold from the mid-point between "immediate" and "delayed" responses, at 148 ms. As before, we established the mode value for the discrimination mid-points from their density plot, which is illustrated in figure 5b. Again the mode yielded a lower value than the median, this time it approximated 121 ms. The distribution showed a wide dispersion of scores around the mode, along with a long tail that ran to the end of the experimental range of 500 ms.

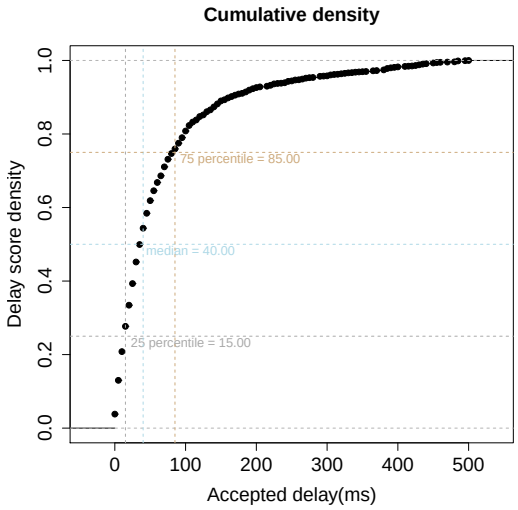


Fig. 3: Individual adjusted delay scores plotted as an empirical cumulative density distribution. The x-axis shows participants' final accepted delay score and the y-axis corresponds to the proportion of scores that fall within defined range.

3.3 Comparison of experimental methodologies

Running a Wilcoxon signed rank-sum test, we settled that the striking difference between the mid-point thresholds established from the two methodologies is statistically significant ($W(49)=1871$, $p<0.001$). While the median for the adjusted delay scores came to 40 ms, the corresponding median for discrimination mid-points was more than 100 ms higher at 148 ms. The same difference was evident for the mode values, established at 30 ms and 121 ms, respectively. Furthermore, scores varied greatly across participants in both experiments. Yet, the wider dispersion of the density plot for discrimination mid-points suggests more individual variation for this task compared to the adjustment task.

4 Discussion

In this work, we have addressed the question on human sensitivity to motor-visual delays. This question has been considered by others before us, in the broad context of human-computer interactions [1], but also for fundemental multisensory processes [4,5,11] and for game-specific scenarios [6,7]. We followed up this body of work with a study that explored isolated motor-visual interactions, focusing on the lower range of delay values. We set out to extend on our ongoing

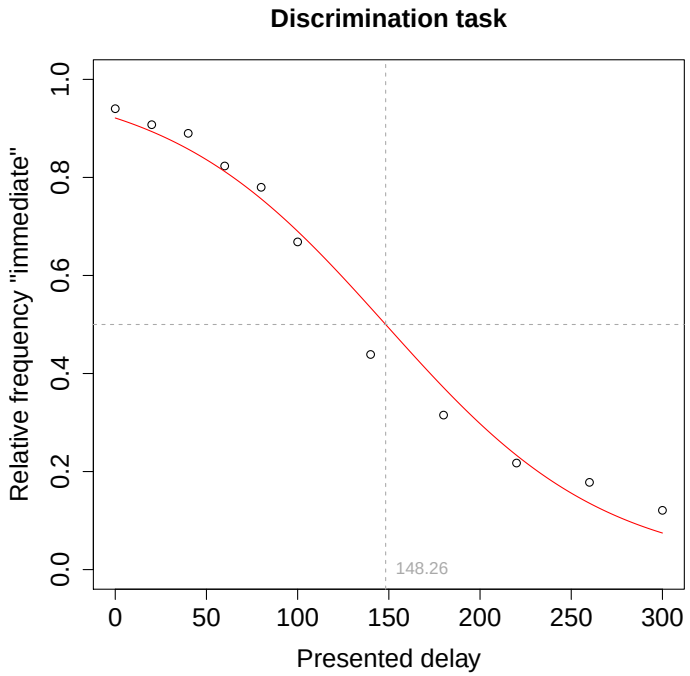
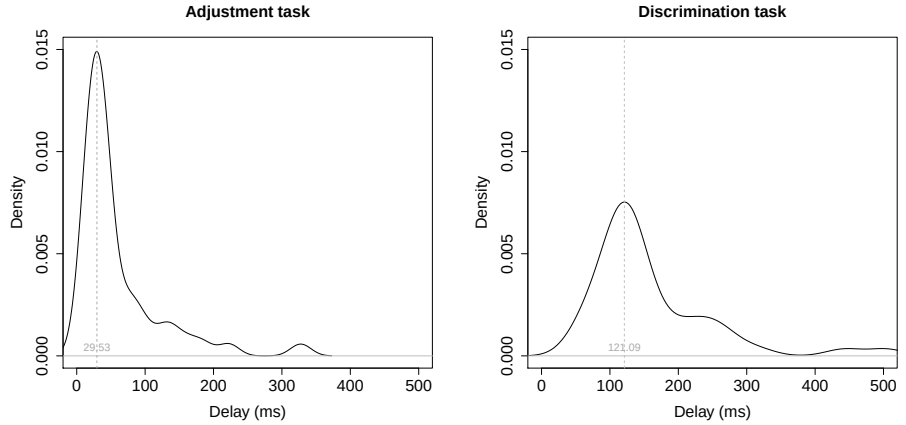


Fig. 4: The best-fitting logistic regression line with the proportion of participants' "immediate" responses plotted as a function of presented motor-visual delays.

innvestigations by introducing dynamic visual stimuli. Furthermore, we sought to assess the generalisibility of results from our adjustment experiment.

In contrast to the higher ranges of motor-visual delays explored by others on this topic [1,4,9], we found that humans are capable of perceiving delays shorter than 100 ms. The thresholds derived from our adjustment experiment indicate that approximately half of the motor-visual adjustments yielded values equal to or below 40 ms, while a quarter of adjustment values fell around or below 15 ms. Keeping in mind that delays related to internal processes are often overlooked, we emphasise that these thresholds are under-estimates of the true delay. By adding the measured system delay to our results (outlined in section 2.3), we include all known sources of delay. Thus we present our most representative thresholds, the 25^{th}, 50^{th} and 75^{th} percentiles at 66 ms, 91 ms and 136 ms.

From our comparison of the two experimental procedures, we found that the discrimination experiment's mid-point was more than 100 ms greater than the adjustment experiment's median. With this marked difference, how do we deem which one is more representative of human delay sensitivity? The answer may lie



(a) Density plot of each participant's median accepted delay in the adjustment task.

(b) Density plot of each participant's midpoint in the discrimination task.

Fig. 5: Subjective delay thresholds presented as density plots

in the context. A short delay can be difficult to perceive with just a single presentation, whereas repeated exposures provide several temporal reference points. Moreover, the discrimination task calls for a simple decision on the precense or absence of delay, whereas the adjustment task rests on the premise that there is a delay present and this should be adjusted down. Hence, the adjustment task dictates engagement from participants and it provides ample opportunity to move past points of uncertainty. On the other hand, the repetitive nature of the adjustment task could also allow participants to adapt a personal strategy to minimise the accepted delay. We observed examples of participants clicking the button quickly and steadily, making the disc bounce back and forth, so they could more easily judge the delay following the click. Relatedly, the repeated adjustments could give way for accepted delay values that fall below the subjective detection thresholds. If a participant manages to find two detection thresholds, one on either side of point zero, they could theoretically turn the wheel midway between the thresholds and accept a value very close to zero. Nevertheless, even when assuming that all pariticpants have adopted such a strategy, the extended range of delay values still fall below the discrimination threshold [13].

Furthermore, the adjustment experiment demonstrates internal validity from the constancy of the derived thresholds. We ran the first experiment with a static visual presentation and a 20 ms temporal resolution on the adjustment wheel. This time we introduced a moving visual presentation and we increased the temporal resolution for adjustments, before we ran the experiment on a new group of participants. Despite the changes, the 25^{th}, 50^{th} and 75^{th} percentiles are virtually identical across the two adjustment experiments. Although there is no

ground for a similar comparison for the discrimination experiment, the density plots in figure 5 show a wider dispersion of subjective thresholds. Accordingly, the task of judging immediacy and delay had our participants accept higher delay values than they could adjust for, and they made their judgements with less consistency. As before, this outcome may be an artefact of strategies adopted by participants. Despite this note of warning, we find it remarkable that several of our participants are capable of manually tuning the motor-visual delays down to values below 100 ms, and do so repeatedly and congruously.

Many human-computer interactions involve more than a single, delayed output. Ongoing tasks that are carried out using a computer tend to involve series of inputs and outputs, similar to our adjustment task. Arguably, the adjustment experiment may be more representative of the scenarios we are interested in. However, what we perceive may not necessarily affect how we perform. In order to understand how motor-visual delays influence not only the conscious experience, but also the interaction itself, we need to evaluate the ability to compensate for delays when performing a task. Thus, we plan to apply the derived range of subjective thresholds, which covers a fairly large sample's sensitiviy to motor-visual delays, to the study of performance on motor-visual tasks. In so doing, we aim to shed more light on which experimental approach provides the most representative estimate of motor-visual temporal sensitivity, and we hope to find out whether perceptible and imperceptible delays can affect performance.

5 Concluding remarks and future work

This study presents findings on human sensitivity to delays between a button-click and a visual presentation, which show that repeated motor-visual interactions can make inherent delays more noticeable. Our results also demonstrate that the dynamics of a visual presentation has little impact on the perceived delay that precedes it. Instead, variations are far greater between individuals. The most sensitive of our participants contributed to establish the 25^{th} percentile at 15 ms, or 66 ms when adding internal system delays. Considering that the median also falls below 100 ms, these outcomes speak in favour of designing interactive systems with very fast responses. Ideally, no user should be able to notice delays during interactions with a computer and the presented thresholds highlight the challenge of meeting these demands.

So far, our investigations have focused on universal thresholds for motor-visual delay sensitivity. Our work shows that this sensitivity varies greatly between individuals, and we wish to explore potential factors that could influence the subjective perception of temporal delays. In particular, we plan to look into earlier encounters with highly time-dependent processes, such as gaming and musical experience. Additionally, some of the work on motor-visual delay ad-

dresses the relevant scenarios directly; for instance, the work of Claypool and colleagues focus on delay in real games [6]. Conversely, we commenced our investigations with simple and isolated motor-visual interactions. In this work, we have extended on our earlier study and added dynamics to the previously static visual presentation. This had very little influence on the derived thresholds for the delay adjustment task. The final step in our on-going work will be to apply motor-visual delays to an interactive task, a simple game, to explore whether task performance is affected by barely noticeable delays.

References

1. S. C. Seow, *Designing and Engineering Time.* Addison-Wesley, 2008.
2. K.-T. Chen and C.-L. Lei, "Are all games equally cloud-gaming-friendly? An electromyographic approach," *Proceedings of the 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pp. 1–6, Nov. 2012.
3. P. Haggard and V. Chambon, "Sense of agency," *Current Biology*, vol. 22, no. 10, pp. R390–R392, May 2012.
4. J. P. Ebert and D. M. Wegner, "Time warp: authorship shapes the perceived timing of actions and events," *Consciousness and Cognition*, vol. 19, no. 1, pp. 481–489, Mar. 2010.
5. M. Rohde, M. Scheller, and M. O. Ernst, "Effects can precede their cause in the sense of agency." *Neuropsychologia*, vol. 65, pp. 191–196, Dec. 2014.
6. M. Claypool and K. Claypool, "Latency and player interaction in online games," vol. 49, no. 11, pp. 40–45, 2006.
7. M. Dick, O. Wellnitz, and L. Wolf, "Analysis of factors affecting players' performance and perception in multiplayer games," in *Proceedings of the 4th ACM SIGCOMM Workshop on Network and System Support for Games (NetGames)*, New York, 2005, pp. 1–7.
8. R. Jota, A. Ng, P. Dietz, and D. Wigdor, "How fast is fast enough? A study of the effects of latency in direct-touch pointing tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Paris, 2013, pp. 2291–2300.
9. D. W. Cunningham, V. A. Billock, and B. H. Tsou, "Sensorimotor adaptation to violations of temporal contiguity," *Psychological Science*, vol. 12, no. 6, pp. 532–535, Nov. 2001.
10. W. Fujisaki and S. Nishida, "Audio-tactile superiority over visuo-tactile and audio-visual combinations in the temporal resolution of synchrony perception." *Experimental Brain Research*, vol. 198, no. 2-3, pp. 245–259, Sep. 2009.
11. J. Heron, J. V. M. Hanson, and D. Whitaker, "Effect before Cause: Supramodal Recalibration of Sensorimotor Timing," *PLoS ONE*, vol. 4, no. 11, p. e7681, 2009.
12. V. Occelli, C. Spence, and M. Zampini, "Audiotactile interactions in temporal perception," *Psychonomic Bulletin & Review*, vol. 18, no. 3, pp. 429–454, Jun. 2011.
13. K. Raaen, R. Eg, and C. Griwodz, "Can gamers detect cloud delay?" in *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games (NetGames)*, vol. 200, Nagoya, 2014.
14. K. Raaen and A. Petlund, "How Much Delay Is There Really in Current Games?" *ACM MMsys*, pp. 2–5, 2015.

Paper VI

Is todays public cloud suited to deploy hardcore realtime services?

Is Today's Public Cloud Suited to Deploy Hardcore Realtime Services?
A CPU Perspective

Kjetil Raaen[1,2,3], Andreas Petlund[2,3], and Pål Halvorsen[2,3]

[1] NITH, Norway
[2] Simula Research Laboratory, Norway
[3] Department of Informatics, University of Oslo, Norway

Abstract. "Cloud computing" is a popular way for application providers to obtain a flexible server and network infrastructure. Providers deploying applications with tight response time requirements such as games, are reluctant to use clouds. An important reason is the lack of real-time guarantees. This paper evaluates the actual, practical soft real-time CPU performance of current cloud services, with a special focus on online games. To perform this evaluation, we created a small benchmark and calibrated it to take a few milliseconds to run (often referred to as a *microbenchmrak*). Repeating this benchmark at a high frequency gives an overview of available resources over time. From the experimental results, we find that public cloud services deliver performance mostly within the requirements of popular online games, where Microsoft Azure Virtual machines give a significantly more stable performance than Amazon EC2.

1 Introduction

A game company planning to deploy an online game will traditionally have to face huge costs for data centre space and bandwidth capacity. Because it is hard to predict a game's popularity before it is launched, the game provider is prone to choose over-provisioning in order to meet potential high demands. Over-provisioning boosts the costs of deployment further. "Cloud computing" is a popular way for other application areas to address this problem. Here, the customer is billed based on the resources used, and can scale resources dynamically. Game companies have, however, been reluctant to place their game servers in a virtualized cloud environment. This reluctance is mainly due to the highly time-dependent nature of games. Many studies have reported that the quality of user experience is highly dependent on latency [3] [4]. For example, Chen et al. [3] examine network latency, but it is safe to assume that delays caused by processing will be perceived identically by players. They find that perceived quality of services in games depends strongly on response time, and variations in response time (jitter) from the server are more important than absolute values. The scale of the delays these papers describe as detrimental to the players lays between 50 and 100 ms. Since providers must allow for normal network latencies

D. an Mey et al. (Eds.): Euro-Par 2013 Workshops, LNCS 8374, pp. 343–352, 2014.
© Springer-Verlag Berlin Heidelberg 2014

in addition to processing time, any significant proportion of this delay incurred by overhead due to the cloud infrastructure should be noted.

The term "Cloud gaming", as currently used, represents "Software as a Service" *(SaaS)* [10], where the software in question is a game. The players will only run a simple thin client, sending player input to the servers, and receiving video and audio from the server. This is the approach followed by OnLive, Gaikai and others. Cloud gaming thus trades the need for local computing power for increased network throughput requirements as well as QoS demands. These services usually use a custom cloud infrastructure optimised for games.

Streaming the game in this way has some disadvantages. First of all, it has significant computing and bandwidth requirements. Many games require the full power of a modern computer to run, and the network traffic is equivalent to video streaming. Using a thin client which simply receives input and outputs video also precludes using any client-side latency-hiding techniques, so games run this way will be more latency sensitive. "Cloud gaming" in this sense is not the topic of this paper.

Conversely, this paper considers using general purpose public clouds to run traditional game servers, serving fully featured game clients. This configuration can be described as online games using cloud infrastructure. Time-dependent applications, like networked games, usually need to be custom-made, and can rarely be built upon typical web- or enterprise frameworks. Hence we need the kind of cloud service known as "Infrastructure as a Service" (IaaS) [10]. This allows the developers to access complete virtual machines, with full access privileges, allowing them to run their fully customised programs in the cloud.

Barker et al.(2010) [2] have investigated many aspects of latency sensitive applications on the Amazon EC2 cloud service. Investigating CPU, disk IO and network performance as well as running an actual game server, they conclude that the variable access to these resources, especially disk and CPU, are enough to degrade performance of latency sensitive applications.

In this paper build on [2] reexamining the CPU stability three years later and add data from one of the competitors that have entered the scene after Amazon: Microsoft Azure. We evaluate the applicability of public infrastructure clouds for use as infrastructure for online games and other highly time-sensitive applications. The work isolates CPU performance, leaving network and other subsystems for further work. We compare response time stability, with respect to processor availability, between cloud services and natively running software, as well as between two different cloud providers. The result is an updated recommendation for services providers.

Other than [2], work on performance in clouds have so far primarily focused on the typical large, bulk jobs of scientific computing. Schad et al. (2010) [9], Ostermann et al. (2010) [7] and [5] have all worked on long term tasks, most finding relevant variation between runs. For the large workloads investigated in this work, transient performance degradations at a millisecond scale, and even stalls will be averaged out and difficult to detect. These issues are, however, highly relevant for game servers, as well as other soft realtime applications.

2 Experiment Design

To evaluate the virtual machines provided by different cloud services for their suitability in real-time applications, we created a benchmark and calibrated it to take a few milliseconds to run. The absolute average values of the benchmark runtime is not important, as we are only interested in the stability of the result. Neither does the load need to behave like any specific realistic load. This evaluation is only concerned with the actual availability of computing resources on a fine-grained time resolution.

With this in mind, the benchmark was designed as a small loop using array lookups and arithmetic, where the array is sized to be significantly larger than the cache size. This allows us to examine the memory subsystem and the CPU performance in isolation, ignoring external I/O. The tight loop of the benchmark is designed as a "worst case" load scenario for a virtual machine. Each machine was tested with a number of threads equal to the number of cores available to the virtual machine. Note that each instance of the benchmark is single-threaded, and will not show "better" results by increasing the number of cores. For these reasons, the most relevant single metric is the *coefficient of variation*. This value is defined as: $c_v = \frac{\sigma}{\mu}$, where σ is the standard deviation and μ is the mean.

In addition to the benchmark code, the utility "mpstat" was run in the background. Mpstat gives us "% steal", a metric that reports how much of the physical CPU is unavailable to the current OS instance because the hypervisor has assigned it to other instances.

Although the paper's main concern is to investigate the variance in execution time for the benchmark, a by-product is that we get an indication of the processing power provided by the different providers/service levels. This may also be of interest to game providers considering cloud deployment.

The benchmark was run on three different instances in Amazon EC2, in the zone "us-east-1d" (Table 1), as well as three different instances in Windows Azure (Table 2).

Table 1. Technical specifications. from amazon [1].

Micro Instance	613 MiB memory Up to 2 EC2 Compute Units (for short periodic bursts) I/O Performance: Low	USD 0.020 per Hour
Medium Instance	3.75 GiB memory 2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit) I/O Performance: Moderate	USD 0.120 per Hour
High-CPU Medium Instance	1.7 GiB of memory 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each) I/O Performance: Moderate	USD 0.145 per Hour

2.1 Short-Term Runs

To compare the different instances, the benchmarks were run for 18 hours each under the configurations shown in tables 1 and 2. To create a baseline for comparing the cloud services with native dedicated hardware, the same benchmark

Table 2. Technical specifications from Microsoft [6]

Extra small instance	768 MiB memory 1 shared core	USD 0.020 per Hour
Small Instance	1.75 GiB memory 1 CPU Core	USD 0.085 per Hour
Medium Instance	3.5 GiB memory 2 CPU Cores	USD 0.160 per Hour

was run a standalone PC (Intel Core 2 Duo CPU E7500 @ 2.93GHz, 4GB RAM) without any visualization, chosen to be approximately the same class of performance as the cloud computers.

2.2 Time Series

Schad et al. [9] suggest that there might be differences in performance of the cloud systems based on when the benchmark is run, either by time of day or day of week. To investigate this, we ran the benchmark described above for three minutes every half hour for a week (the week of 27th of May, 2013). Between each run, we stopped the instance, and started it again before the next run. According to [9], this should allow us to get a different physical machine every time.

3 Evaluation

3.1 Reference System

The reference system (figure 1) shows the baseline of consistency we can expect from this benchmark while running natively. Deviations from this baseline can be assumed to result from the hypervisor or other aspects of the cloud system. As we can observe, the benchmark here reports extremely consistent results. The c_v is 0.02.

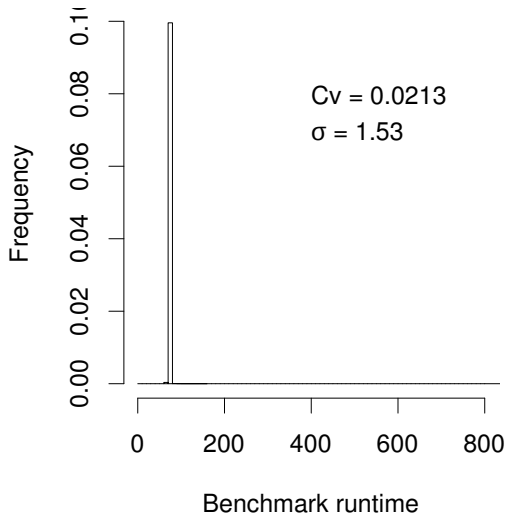
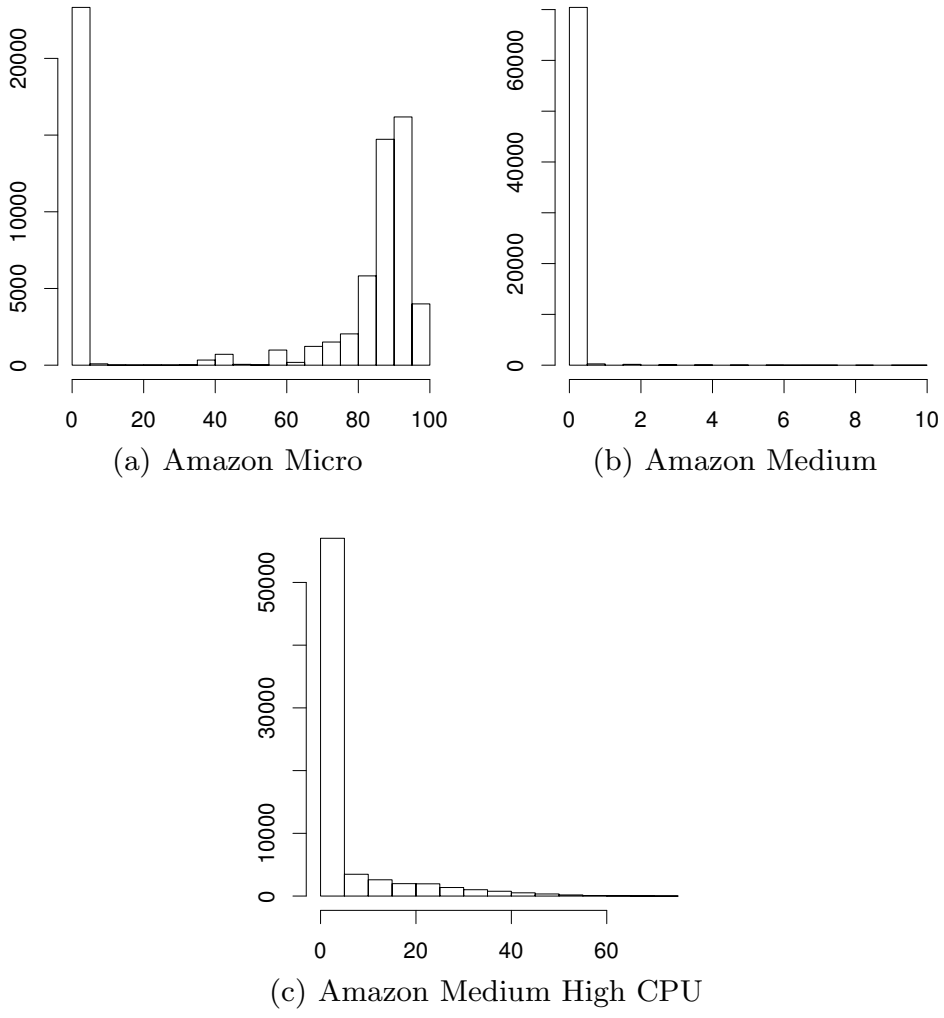


Fig. 1. Benchmark results from reference system

3.2 Amazon EC2

CPU Steal. This metric is only sampled once a second, and so has a low resolution, but with sufficient samples we build up a picture of how much CPU is allocated to each instance, as well as the distribution over time.



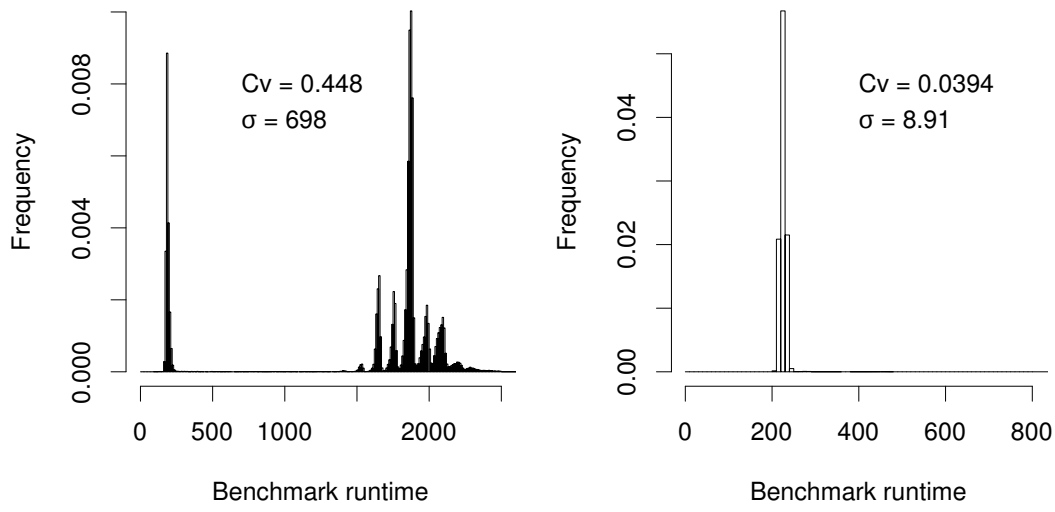
(a) Amazon Micro (b) Amazon Medium

(c) Amazon Medium High CPU

Fig. 2. CPU Steal histograms. Samples every second.

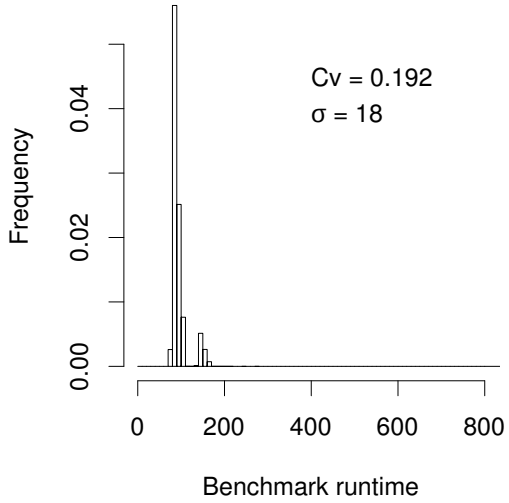
The first thing to note in this data (figure 2) is the fact that all the instance types have a significant number of samples where the current instance has full use of the CPU. These situations will pull the average CPU availability significantly up, allowing the provider to fulfil their obligations despite other samples with much less available resources.

The next point standing out is the "medium" instance. According to the pricing information, the available CPU resources on this instance type equals exactly one core of the current hardware. This seems to create a situation where our instance gets full use of the core most of the time, with very little steal. Increasing the available CPU power to the "high cpu" instance adds another



(a) EC2 Micro

(b) EC2 Medium



(c) EC2 Medium High CPU

Fig. 3. Amazon EC2: Histogram of benchmark runtimes. Note different scale on the axis of 3(a).

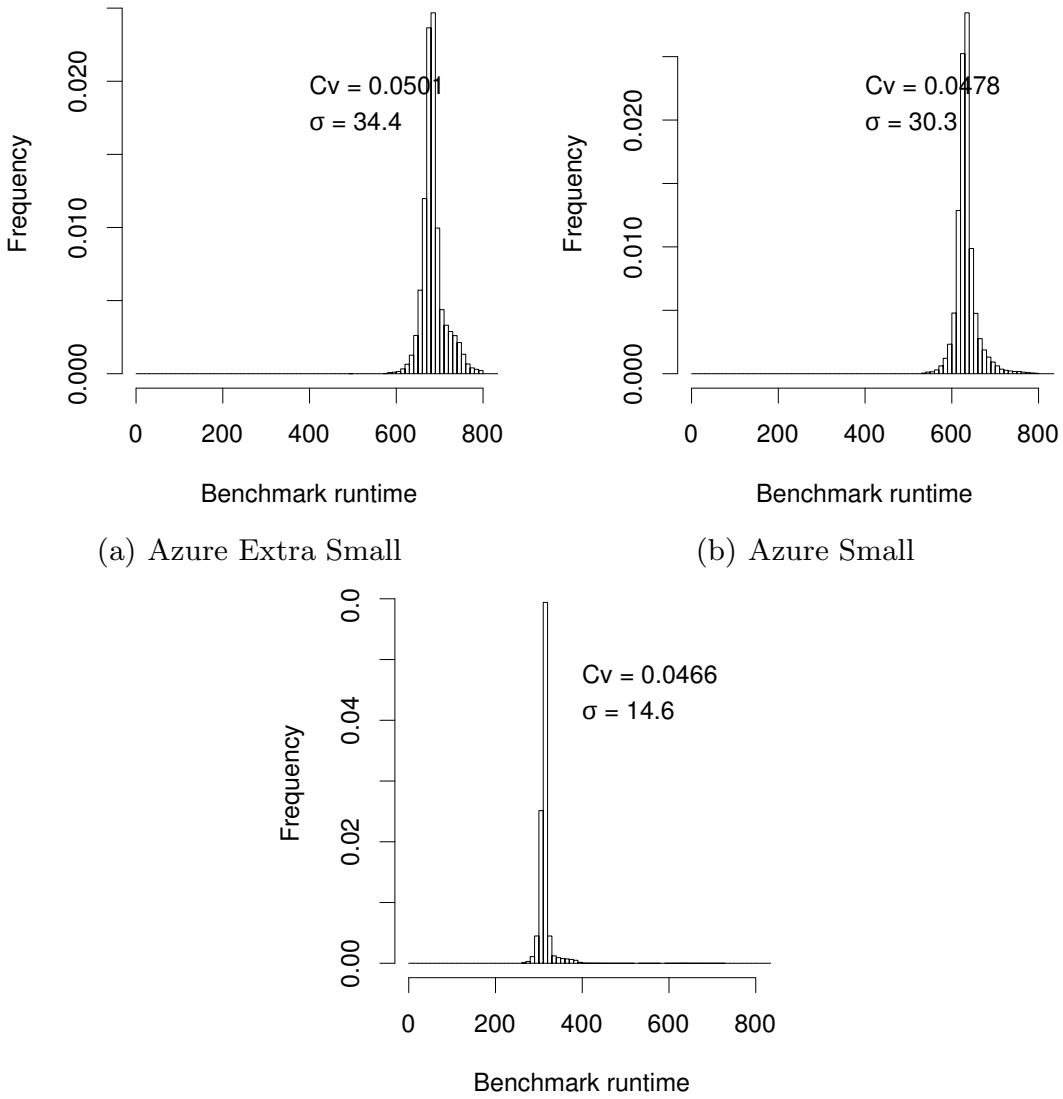
core, but since the instance on average only has access to part of this core, the cpu steal value increases again.

Benchmark Runtime. These results (figure 3) show several interesting values. First of all, for both the "micro" as well as the "medium, high-CPU" instance, there are two clear peaks. This matches the CPU-steal data, where the low values represent the situations where our instance got full use of the CPU, and the high run-times are the result of periods of higher CPU-steal. Again we see the advantage of the "medium" instance, in that it has a much more stable runtime, with c_v equal to 0.04. This more stable instance type depends on the fact that it allocates exactly one core to the instance. As the processing power granted each instance type is defined by an abstract, hardware independent metric, clients have no guarantee that this situation will continue indefinitely. Rather, when

the underlying hardware is upgraded, it is very likely that each core will provide more power, and the currently stable instance type will become unstable.

3.3 Microsoft Azure

CPU Steal. On the "Microsoft Azure Virtual Machine" cloud, the operating system always reports 0 CPU steal, as if it was running on its own dedicated machine. This implies that the hypervisor hides these details from the operating system, or that our instance actually has a dedicated CPU resource.



(a) Azure Extra Small

(b) Azure Small

(c) Azure Medium

Fig. 4. Microsoft Azure: Histograms of benchmark runtimes

Benchmark Runtime. Compared to the Amazon case, access to the CPU is significantly more stable in the Azure cloud (figure 4). Regardless of instance

type, the runtime of the benchmark is almost completely predictable and stable, c_v is 0.05 for all instance types. The deviation is however twice that of the reference case. Depending on the exact level of time-sensitivity of the load, and its computational cost, this could be acceptable or not. The single-threaded nature of the benchmark explains why the "small" and "medium" show almost identical results.

3.4 Time Series

Figure 5 shows the results of running the benchmark repeatedly over a week on an EC2 Medium Highcpu instance. The two bands of performance are visible throughout the week, and there is no clear difference based on weekday or time of day.

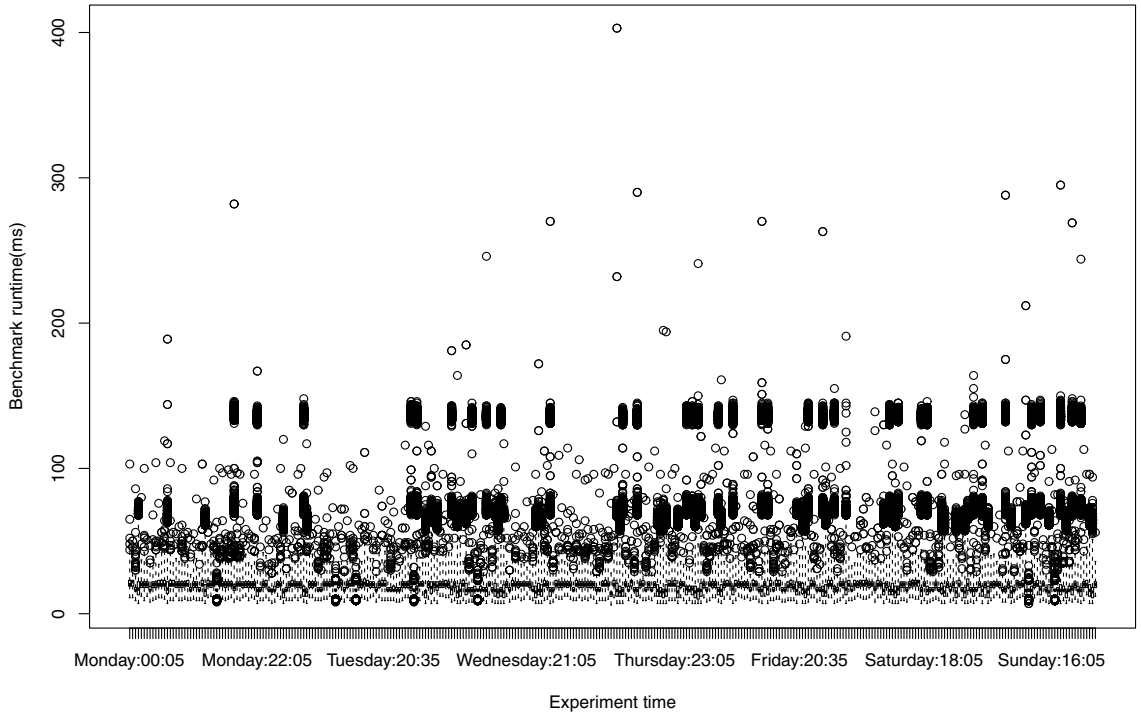


Fig. 5. Time series of Amazon EC2 Medium High CPU

3.5 Overview

Putting it all together (figure 6) we see the quite clear differences between the providers. The low-end instance from Amazon has variations far above the acceptable level for any time-sensitive application. Interestingly the "Amazon EC2 Medium High CPU" is also quite unpredictable, though this configuration is sold as higher performing than the "Medium" type. Among the Amazon offerings examined only the "Medium" instance type is near acceptable. From Microsoft Azure, all instance types are reasonably stable in performance. All show variations above our native machine reference. The significance of these variations depend on the requirements of the application.

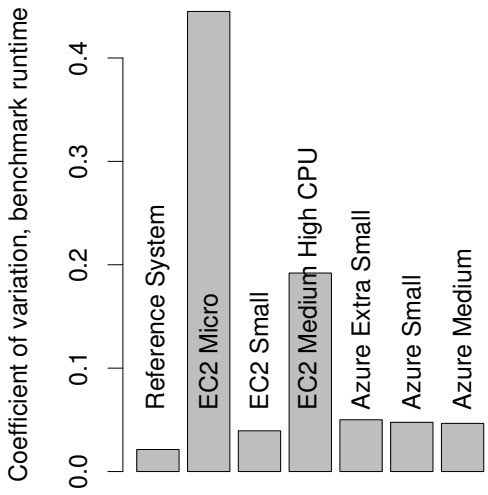


Fig. 6. Summary of benchmarks for the different systems

4 Conclusion

From these simple tests, it seems that the conclusions about CPU usage from [2] still hold for Amazon EC2. For the more time sensitive class of games [4], these delays can add up with network latency to unacceptable levels. Even for less sensitive applications there are some caveats. We do get rare samples where the CPU seems to have stalled for multiple seconds which could discourage users. Microsoft Azure Virtual machines seem to give more stable performance than Amazon EC2, without the extreme latency peaks. Utilizing these results can allow providers of interactive online services to be much more agile and react faster to changes in demand.

For future work on clouds and other virtual machines, we found that the c_v metric of multiple runs of a small, simple benchmark is a good starting point for measuring the consistency of computing resources.

Based on our measurements, access to processors is sufficiently stable on today's IaaS cloud systems to allow for real-time services to be deployed, assuming the servers are run on the right instance types.

Amazon EC2 is better at the higher service levels, although variance at all levels is larger than for Azure. An important element to keep in mind for Amazon EC2 is that migration to a different instance is necessary if a change in service level is needed.

Azure cloud shows less variance in processor availability results than Amazon. Although you get less processor speed for the same price, it is more reliable for planning stability, reducing the need for a specialized service for managing changes between service levels. If a change in service level is needed, though, Azure allows for dynamic changes without the need to migrate the virtual machine.

In this paper we have evaluated Amazon EC2 and Microsoft Azure Cloud. To give wider recommendations we are planning similar experiments using Rackspace and Google Cloud. Google Cloud currently requires the customer to go through

an application process for access. We did not get any reply from Google on our request. Rackspace have not been included due to time and space requirements.

To provide a complete view of the conditions for running real-time services on IaaS instances, we need to extend our result with networking measurements, extending [2].

In order to minimize the experienced processing time variance, monitoring the experienced processing time c_v and changing the service level based on the results should be explored. This will require different methods in Amazon EC2 and Azure, but should be a useful tool for game service providers that are concerned about processing jitter.

With more detailed knowledge of how virtual machines in the cloud respond, it would be relevant to experiment with more realistic, preferably highly parallel workloads such as LEARS [8].

References

1. Amazon. Amazon EC2 Instance Types (2013), `http://aws.amazon.com/ec2/instance-types/`
2. Barker, S.K., Shenoy, P.: Empirical evaluation of latency-sensitive application performance in the cloud. In: Proceedings of ACM SIGMM on Multimedia Systems, MMSys 2010, pp. 35–46. ACM, New York (2010)
3. Chen, K.-T., Huang, P., Wang, G.-S., Huang, C.-Y., Lei, C.L.: On the Sensitivity of Online Game Playing Time to Network QoS. In: Proceedings of IEEE INFOCOM 2006(2006)
4. Claypool, M., Claypool, K.: Latency Can Kill: Precision and Deadline in Online Games. In: ACM Multimedia Systems Conference (2010)
5. El-Khamra, Y., Kim, H., Jha, S., Parashar, M.: Exploring the Performance Fluctuations of HPC Workloads on Clouds. In: 2010 IEEE Cloud Computing Technology and Science, pp. 383–387 (November 2010)
6. Microsoft. Microsoft Azure Pricing Details (2013), `http://www.windowsazure.com/en-us/pricing/details/`
7. Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D.: A performance analysis of EC2 cloud computing services for scientific computing. In: Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., Epema, D. (eds.) Cloud Computing. LNICST, vol. 34, pp. 115–131. Springer, Heidelberg (2010)
8. Raaen, K., Espeland, H.: LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition. In: Parallel Processing ..., pp. 382–389. IEEE (September 2012)
9. Schad, J., Dittrich, J., Quiané-Ruiz, J.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. Proceedings of the VLDB ... 3(1) (2010)
10. Vaquero, L.M., Rodero-merino, L., Caceres, J., Lindner, M.: A Break in the Clouds: Towards a Cloud Definition. Computer Communication Review 39(1), 50–55 (2009)

Paper VII

LEARS: A Lockless, relaxed-atomicity state model for parallel execution of a game server partition

2012 41st International Conference on Parallel Processing Workshops

LEARS: A Lockless, Relaxed-Atomicity State Model for Parallel Execution of a Game Server Partition

Kjetil Raaen, Håvard Espeland, Håkon K. Stensland, Andreas Petlund, Pål Halvorsen, Carsten Griwodz
NITH, Norway Simula Research Laboratory, Norway IFI, University of Oslo, Norway
Email: raakje@nith.no, {haavares, haakonks, apetlund, paalh, griff}@ifi.uio.no

***Abstract*—Supporting thousands of interacting players in a virtual world poses huge challenges with respect to processing. Existing work that addresses the challenge utilizes a variety of spatial partitioning algorithms to distribute the load. If, however, a large number of players needs to interact tightly across an area of the game world, spatial partitioning cannot subdivide this area without incurring massive communication costs, latency or inconsistency. It is a major challenge of game engines to scale such areas to the largest number of players possible; in a deviation from earlier thinking, parallelism on multi-core architectures is applied to increase scalability. In this paper, we evaluate the design and implementation of our game server architecture, called LEARS, which allows for lock-free parallel processing of a single spatial partition by considering every game cycle an atomic tick. Our prototype is evaluated using traces from live game sessions where we measure the server response time for all objects that need timely updates. We also measure how the response time for the multi-threaded implementation varies with the number of threads used. Our results show that the challenge of scaling up a game-server can be an embarrassingly parallel problem.**

I. Introduction

Over the last decade, online multi-player gaming has experienced an amazing growth. Providers of the popular online games must deliver a reliable service to thousands of concurrent players meeting strict processing deadlines in order for the players to have an acceptable quality of experience (QoE).

One major goal for large game providers is to support as many concurrent players in a game-world as possible while preserving the strict latency requirements in order for the players to have an acceptable quality of experience (QoE). Load distribution in these systems is typically achieved by partitioning game-worlds into areas-of-interest to minimize message passing between players and to allow the game-world to be divided between servers. Load balancing is usually completely static, where each area has dedicated hardware. This approach is, however, limited by the distribution of players in the game-world, and the problem is that the distribution of players is heavy-tailed with about 30% of players in 1% of the game area [5]. To handle the most popular areas of the game world without reducing the maximum interaction distance for players, individual spatial partitions can not be serial. An MMO-server will experience the most CPU load while the players experience the most "action". Hence, the worst case scenario for the server is when a large proportion of the players gather in a small area for high intensity gameplay.

In such scenarios, the important metric for online multi-player games is latency. Claypool et. al. [7] classify different types of games and conclude that for first person shooter (FPS) and racing games, the threshold for an acceptable latency is 100ms. For other classes of networked games, like real-time strategy (RTS) and massively multi-player online games (MMOGs) players tolerate somewhat higher delays, but there are still strict latency requirements in order to provide a good QoE. The accumulated latency of network transmission, server processing and client processing adds up to the latencies that the user is experiencing, and reducing any of these latencies improves the users' experience.

The traditional design of massively multi-player game servers rely on *sharding* for further load distribution when too many players visit the same place simultaneously. Sharding involves making a new copy of an area of a game, where players in different copies are unable to interact. This approach eliminates most requirements for communication between the processes running individual shards. An example of such a design can be found in [6].

The industry is now experimenting with implementations that allow for a greater level of parallelization. One known example is Eve Online [8] where they avoid *sharding* and allow all players to potentially interact. Large-scale interactions in Eve Online are handled through an optimized database. On the local scale, however, the server is not parallel, and performance is extremely limited when too many players congregate in one area. With LEARS, we take this approach even further and focus on how many players that can be handled in a single segment of the game world. We present a model that allows for better resource utilization of multi-processor, game server systems which should not replace spatial partitioning techniques for work distribution, but rather complement them to improve on their limitations. Furthermore, a real prototype game is used for evaluation where captured traces are used to generate server load. We compare multi-threaded and single-threaded implementations in order to measure the overhead of parallelizing the implementation and showing the experienced benefits of parallelization. The change in responsiveness of different implementations with increased load on the server is

1530-2016/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPPW.2012.55

CPS Conference Publishing Services

studied, and we discuss how generic elements of this game design impact the performance on our chosen platform of implementation.

Our results indicate that it is possible to design an "embarrassingly parallel" game server. We also observe that the implementation is able to handle a quadratic increase of in-server communication when many players interact in a game-world hotspot.

The rest of the paper is organized as follows: In section II, we describe the basic idea of LEARS, before we present the design and implementation of the prototype in section III. We evaluate our prototype in section IV and discuss our idea in section V. In section VI, we put our idea in the context of other existing work. Finally, we summarize and conclude the paper in section VII and give directions for further work in section VIII.

II. LEARS: The Basic Idea

Traditionally, game servers have been implemented much like game clients. They are based around a main loop, which updates every active element in the game. These elements include for example player characters, non-player characters and projectiles. The simulated world has a list of all the active elements in the game and typically calls an "update" method on each element. The simulated time is kept constant throughout each iteration of the loop, so that all elements get updates at the same points in simulated time. This point in time is referred to as a *tick*. Using this method, the active element performs all its actions for the tick. Since only one element updates at a time, all actions can be performed directly. The character reads input from the network, performs updates on itself according to the input, and updates other elements with the results of its actions.

LEARS is a game server model with support for lockless, relaxed-atomicity state-parallel execution. The main concept is to split the game server executable into lightweight threads at the finest possible granularity. Each update of every player character, AI opponent and projectile runs as an independent work unit.

White et al. [15] describe a model they call a *state-effect pattern*. Based on the observation that changes in a large, actor-based simulation are happening *simultaneously*, they separate read and write operations. Read operations work on a consistent previous state, and all write operations are batched and executed to produce the state for the next tick. This means that the ordering of events scheduled to execute at a tick does not need to be considered or enforced. For the design in this paper, we additionally remove the requirement for batching of write operations, allowing these to happen anytime during the tick. The rationale for this relaxation is found in the way traditional game servers work. In the traditional single-threaded main-loop approach, every update is allowed to change any part of the simulation state at any time. In such a scenario the state at a given time is a combination of values from two different points in time, current and previous, exactly the same situation that occurs in the design presented here.

The second relaxation relates to the atomicity of game state updates. The fine granularity creates a need for significant communication between threads to avoid problematic lock contentions. Systems where elements can only update their own state and read any state without locking [1] do obviously not work in all cases. However, game servers are not accurate simulators, and again, depending on the game design, some (internal) errors are acceptable without violating game state consistency. Consider the following example: Character A moves while character B attacks. If only the X coordinate of character A is updated at the point in time when the attack is executed, the attack sees character A at a position with the new X coordinate and the old Y coordinate. This position is within the accuracy of the simulation which in any case is no better than the distance an object can move within one tick.

On the other hand, for actions where a margin of error is not acceptable, transactions can be used keeping the object's state internally consistent. However, locking the state is expensive. Fortunately, most common game actions do not require transactions, an observation that we take advantage of in LEARS.

These two relaxations allow actions to be performed on game objects in any order without global locking. It can be implemented using message passing between threads and retains consistency for most game actions. This includes actions such as moving, shooting, spells and so forth. Consider player A shooting at player B: A subtracts her ammunition state, and send bullets in B's general direction by spawning bullet objects. The bullet objects runs as independent work units, and if one of them hits player B, it sends a message to player B. When reading this message, player B subtracts his health and sends a message to player A if it reaches zero. Player A then updates her statistics when she receives player B's message. This series of events can be time critical at certain points. The most important point is where the decision is made if the bullet hits player B. If player B is moving, the order of updates can be critical in deciding if the bullet hits or misses. In the case where the bullet moves first, the player does not get a chance to move out of the way. This inconsistency is however not a product of the LEARS approach. Game servers in general insert active items into their loops in an arbitrary fashion, and there is no rule to state which order is "correct".

The end result of our proposed design philosophy is that there is no synchronization in the server under normal running conditions. Since there are cases where transactions are required, they can be implemented outside the LEARS event handler running as transactions requiring locking. In the rest of the paper, we consider a practical implementation of LEARS, and evaluate its performance and scalability.

III. Design and Implementation

In our experimental prototype implementation of the LEARS concept, the parallel approach is realized using thread pools and blocking queues.

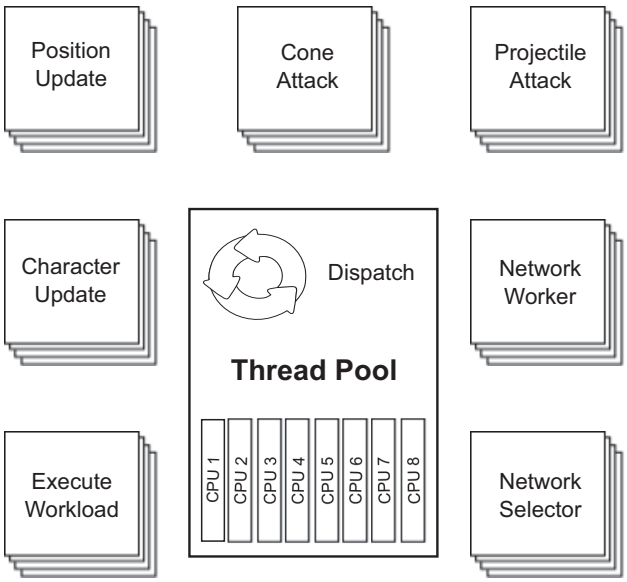


Figure 1. Design of the Game Server

A. Thread pool

Creation and deletion of threads incur large overheads, and context switching is an expensive operation. These overheads constrain how a system can be designed, i.e., threads should be kept as long as possible, and the number of threads should not grow unbounded. We use a *thread pool* pattern to work around these constraints, and a thread pool executor (the Java `ThreadPoolExecutor` class) to maintain the pool of threads and a queue of tasks. When a thread is available, the executor picks a task from the queue and executes it. The thread pool system itself is not preemptive, so the thread runs each task until it is done. This means that in contrast to normal threading, each task should be as small as possible, i.e., larger units of work should be split up into several sub-tasks.

The thread pool is a good way to balance the number of threads when the work is split into extremely small units. When an active element is created in the virtual world, it is scheduled for execution by the thread pool executor, and the active element updates its state exactly as in the single threaded case. Furthermore, our thread pool supports the concept of delayed execution. This means that tasks can be put into the work queue for execution at a time specified in the future. When the task is finished for one time slot, it can reschedule itself for the next slot, delayed by a specified time. This allows active elements to have any lifetime from one-shot executions to the duration of the program. It also allows different elements to be updated at different rates depending on the requirements of the game developer.

All work is executed by the same thread pool, including the slower I/O operations. This is a consistent and clear approach, but it does mean that game updates could be stuck waiting for I/O if there are not enough threads available.

B. Blocking queues

The thread pool executor used as described above does not constrain which tasks are executed in parallel. All systems elements must therefore allow any of the other elements to execute concurrently.

To enable a fast communication between threads with shared memory (and caches), we use *blocking queues*, using the Java `BlockingQueue` class, which implements queues that are synchronized separately at each end. This means that elements can be removed from and added to the queue simultaneously, and since each of these operations are extremely fast, the probability of blocking is low. In the scenario analysed here, all active elements can potentially communicate with all others. Thus, these queues allow information to be passed between active objects. Each active object that can be influenced by others has a blocking queue of messages. During its update, it reads and processes the pending messages from its queue. Messages are processed in the order they were put in the queue. Other active elements put messages in the queue to be processed when they need to change the state of other elements in the game.

Messages in the queues can only contain relative information, and not absolute values. This restriction ensures that the change is always based on updated data. For example, if a projectile needs to tell a player character that it took damage, it should only inform the player character about the amount of damage, not the new health total. Since all changes are put in the queue, and the entire queue is processed by the same work unit, all updates are based on up-to-date data.

C. Our implementation

To demonstrate LEARS, we have implemented a prototype game containing all the basic elements of a full MMOG with the exception of persistent state. The basic architecture of the game server is described in figure 1. The thread pool size can be configured, and will execute the different workloads on the CPU cores. The workloads include processing of network messages, moving computer controlled elements (in this prototype only projectiles) checking for collisions and hits and sending outgoing network messages.

Persistent state do introduce some complications, but as database transactions are often not time critical and can usually be scheduled outside peak load situations, we leave this to future work.

In the game, each player controls a small circle ("the character") with an indicator for which direction they are heading (see figure 2). The characters are moved around by pressing keyboard buttons. They also have two types of attack, i.e., one projectile and one instant area of effect attack. Both attacks are aimed straight ahead. If an attack hits another player character, the attacker gets a positive point, and the character that was hit gets a negative point. The game provides examples of all the elements of the design described above:

- The player character is a long lifetime active object. It processes messages from clients, updates states and potentially produces other active objects (attacks). In

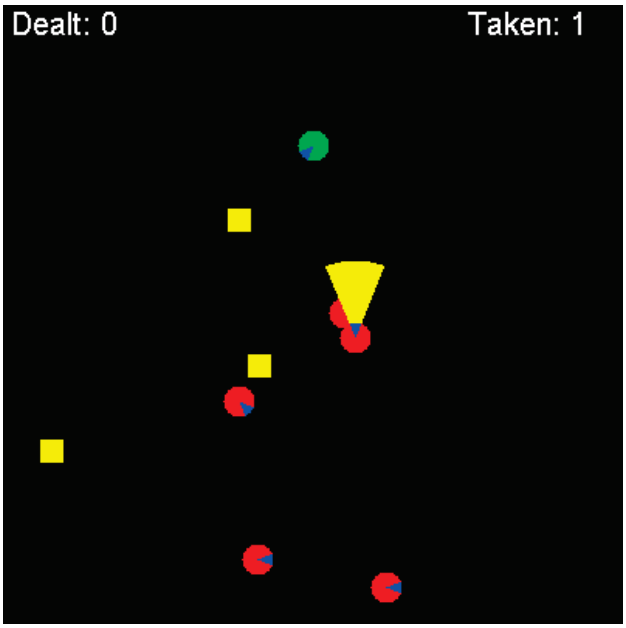


Figure 2. Screen shot of a game with six players.

addition to position, which all objects have, the player also has information about how many times it has been hit and how many times it has hit others. The player character also has a message queue to receive messages from other active objects. At the end of its update, it enqueues itself for the next update unless the client it represents has disconnected.

- The frontal cone attack is a one shot task that finds player characters in its designated area and sends messages to those hit so they can update their counters, as well as back to the attacking player informing about how many were hit.
- The projectile is a short lifetime object that moves in the world, checks if it has hit anything and reschedules itself for another update, unless it has hit something or ran to the end of its range. The projectile can only hit one target.

To simulate an MMORPG workload that grow linearly with number of players, especially collision checks with the ground and other static objects, we have included a synthetic load which emulates collision detection with a high-resolution terrain mesh. The synthetic load ensures that the cache is regularly flushed to enhance the realism of our game server prototype compared to a large-scale game server.

The game used in these experiments is simple, but it contains examples of all elements typically available in the action based parts of a typical MMO-like game.

The system described in this paper is implemented in Java. This programming language has strong support for multi-threading and has well-tested implementations of all the required components. The absolute values resulting from these experiments depend strongly on the complexity of the game, as a more complex game would require more processing. In addition, the absolute values depend on the runtime environment, especially the server hardware, and the choice of programming language also influence absolute results from the experiments. However, the focus of this paper is the relative results, as we are interested in comparing scalability of the multi-threaded solution with a single-threaded approach and whether the multi-threaded implementation can handle the quadratic increase in traffic as new players join.

IV. Evaluation

To have a realistic behavior of the game clients, the game was run with 5 human players playing the game with a game update frequency of 10 Hz. The network input to the server from this session was recorded with a timestamp for each message. The recorded game interactions were then played back multiple times in parallel to simulate a large number of clients. To ensure that client performance is not a bottleneck, the simulated clients were distributed among multiple physical machines. Furthermore, as an average client generates 2.6 kbps network traffic, the 1 Gbps local network interface that was used for the experiments did not limit the performance. The game server was run on a server machine containing 4 Dual-Core AMD Opteron 8218 (2600 MHz) with 16 GB RAM. To ensure comparable numbers, the server was taken down between each test run.

A. Response latency

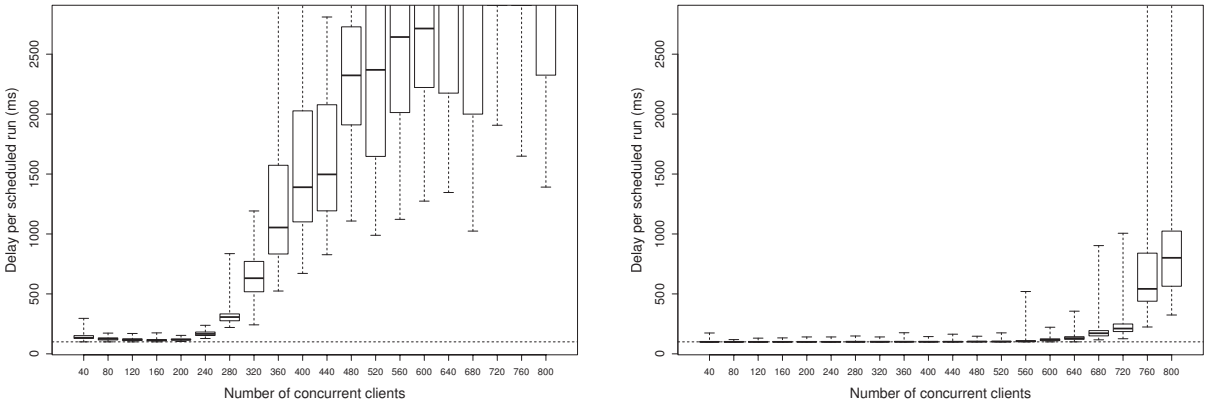
The most important performance metric for client-server games is response latency from the server. From a player perspective, latency is only visible when it exceeds a certain threshold. Individual peaks in response time are obvious to the players, and will have the most impact on the Quality of Experience, hence we focus on peak values as well as averages in the evaluation.

The experiments were run with client numbers ranging from 40 to 800 in increments of 40, where the goal is to keep the latencies close to the 100 ms QoE threshold for FPS games [7]. Figure 3 shows a box-plot of the response time statistics from these experiments. All experiments used a pool of 48 worker threads and distributed the network connections across 8 IP ports.

From these plots, we can see that the single-threaded implementation is struggling to support 280 players at an average latency close to 100 ms. The median response time is 299 ms, and it already has extreme values all the way to 860 ms, exceeding the threshold for a good QoE. The multi-threaded server, on the other hand, is handling the players well up to 640 players where we are getting samples above 1 second, and the median is at 149 ms.

These statistics are somewhat influenced by the fact that the number of samples is proportional to the update frequency. This means that long update cycles to a certain degree get artificially lower weight.

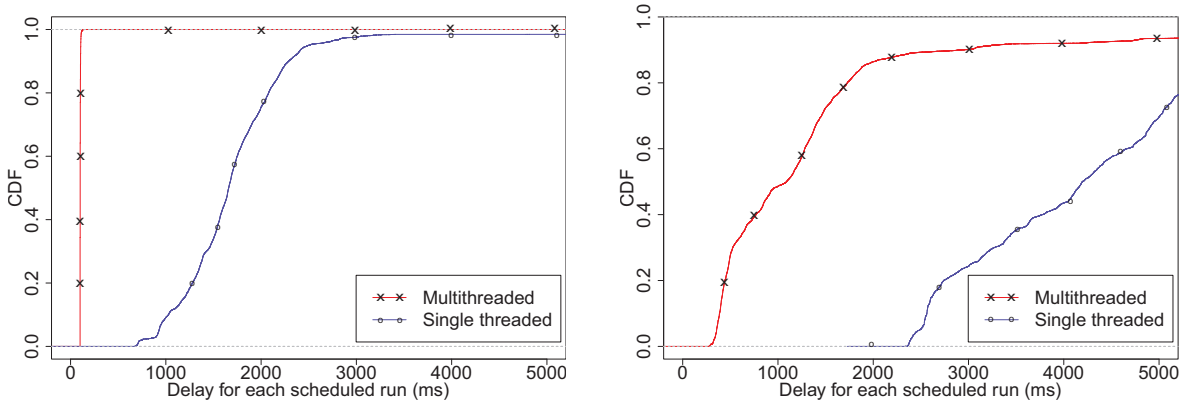
Figure 4 shows details of two interesting cases. In figure 4(a), the single-threaded server is missing all its deadlines with 400 concurrent players, while the multi-threaded version



(a) Single-threaded server

(b) Multi-threaded server

Figure 3. Response time for single- and multi-threaded servers (dotted line is the 100 ms threshold).



(a) 400 concurrent clients

(b) 800 concurrent clients

Figure 4. CDF of response time for single- and multi-threaded servers with 400 and 800 concurrent clients.

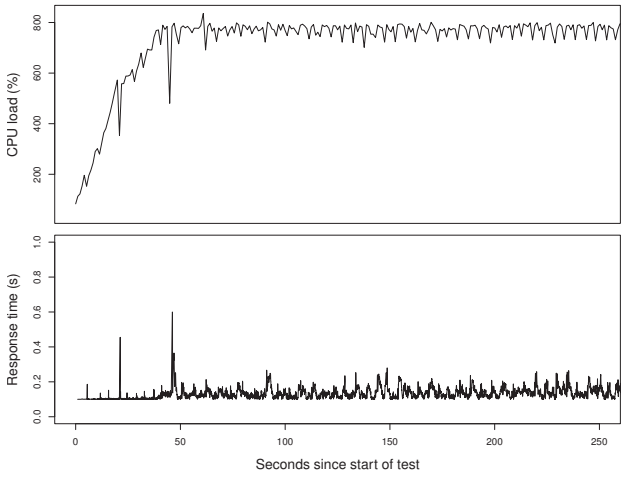


Figure 5. CPU load and response time for 620 concurrent clients on the multi-threaded server.

is processing almost everything on time. At 800 players (figure 4(b)), the outliers are going much further for both cases. Here, even the multi-threaded implementation is struggling to keep up, though it is still handling the load significantly better than the single-threaded version, which is generally completely unplayable.

B. Resource consumption

We have investigated the resource consumption when players connect to the multhreaded server as shown in figure 5. We present the results for 620 players, as this is the highest number of simultaneous players that server handles before significant degradation in performance, as shown in figure 3(b). The mean response time is 133 ms, above the ideal delay of 100 ms. Still, the server is able to keep the update rate smooth, without significant spikes. The CPU utilization grows while the clients are logging on, then stabilizes at an almost full CPU utilization for the rest of the run. The two spikes in response time happen while new players log in to the server at a very fast rate (30 clients pr. second). Receiving a new player requires a lock in

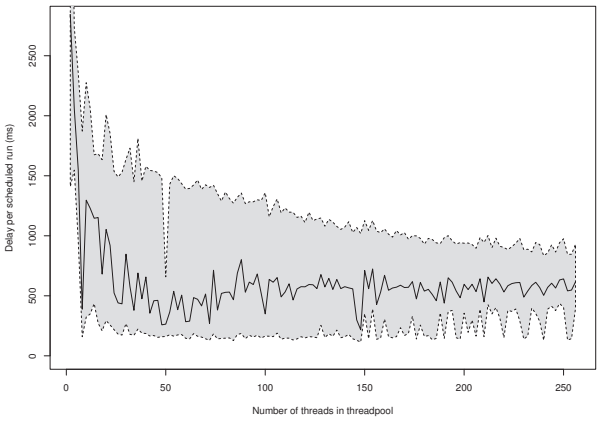


Figure 6. Response time for 700 concurrent clients on using varying number of threads. Shaded area from 5 to 95 percentiles.

the server, hence this operation is, to a certain degree, serial.

C. Effects of thread-pool size

To investigate the effects of the number of threads in the threadpool, we performed an experiment where we kept the number of clients constant while varying the number of threads in the pool. 700 clients were chosen, as this number slightly overloads the server. The number of threads in the pool was increased in increments of 2 from 2 to 256. In figure 6, we see clearly that the system utilizes more than 4 cores efficiently, as the 4 thread version shows significantly higher response times. At one thread per core or more, the numbers are relatively stable, with a tendency towards more consistent low response times with more available threads, to about 40 threads. This could mean that threads are occasionally waiting for I/O operations. Since thread pools are not pre-emptive, such situations would lead to one core going idle if there are no other available threads. Too many threads, on the other hand, could lead to excessive context switch overhead. The results show that the average is slowly increasing after about 50 threads, though the 95-percentile is still decreasing with increased number of threads, up to about 100. From then on the best case is worsening again most likely due to context switching overhead.

A game developer needs to consider this trade-off when tuning the parameters for a specific game.

V. Discussion

Most approaches to multi-threaded game server implementations in the literature (e.g., [1]) use some form of *spatial partitioning* to lock parts of the game world while allowing separate parts to run in parallel. Spatial partitioning is also used in other situations to limit workload. The number of players that game designers can allow in one area in a game server is limited by the worst-case scenario. The worst case scenario for a spatially partitioned game world is when everybody move to the same point, where the spatial partitioning still ends up with everybody in the same partition regardless of granularity. This paper investigates an orthogonal and complementary approach which tries to increase the maximum number of users in the worst case scenario where all players can see each other at all times. Thus, spatial partitioning could be added to further scale the game server.

Experiments using multiple instances of a single-threaded server are not performed, as having clients distribueted acrosss multiple servers would mean partitioning the clients in areas where they can not interact, making numbers from such a scenario incomparable to the multithreaded solutions.

The LEARS approach does have *limitations* and is for example not suitable if the outcome of a message put restrictions on an object's state. This is mainly a game design issue, but situations such as trades can be accommodated by doing full transactions. The following example where two players trade illustrates the problem: Player A sends a message to player B where he proposes to buy her sword for X units. After this is sent, player C steals player A's money, and player A is unable to pay player B should the request go through. This is only a problem for trades *within* a single game tick where the result of a message to another object puts a constraint on the original sender, and can be solved by means such as putting the money in escrow until the trade has been resolved, or by doing a transaction outside of LEARS (such as in a database). Moreover, the design also adds some overhead in that the code is somewhat more complex, i.e., all communication between elements in the system needs to go through message queues. The same issue will also create some runtime overhead, but our results still demonstrate a significant benefit in terms of the supported number of clients.

Tasks in a thread pool can not be pre-empted, but the threads used for execution can. This distinction creates an interesting look into the performance trade-off of pre-emption. If the number of threads in the threadpool is equal to the number of CPU cores, we have a fully cooperative multitasking system. Increasing the number of threads allow for more pre-emption, but introduces context-switching overhead.

VI. Related Work

At Netgames 2011 [12], we presented a demo with a preliminary version of LEARS. Significant research has been done on how to optimize game server architectures for online games, both MMOGs and smaller-scale games. In this section, we summarize some of the most important findings from related research in this field. For example, "Red Dwarf", the community-based successor to "Project Darkstar" by Sun Microsystems [13], is a good example of a parallel approach to game server design. Here, response time is considered one of the most important metrics for game server performance, and suggests a parallel approach for scaling. The described system uses transactions for all updates to world state, including player position. This differs from LEARS, which investigates the case for common actions where atomicity of transactions is not necessary.

Work has also been done on scaling games by looking at the optimization as a data management problem. The authors in [14] have developed a highly expressive scripting language called SGL that provides game developers a data-driven AI scheme for non-player characters. By using query processing and indexing techniques, they can efficiently scale to a large number of non-player objects in games. This group also introduces the concept *state-effect pattern* in [15], which we extend in this paper. They test this and other parallel concepts using a simulated actor interaction model, in contrast to this paper which evaluates a running prototype of a working games under realistic conditions.

Moreover, Cai et al. [4] present a scalable architecture for supporting large-scale interactive Internet games. Their approach divides the game world into multiple partitions and assigns each partition to a server. The issues with this solution is that the architecture of the game server is still a limiting factor in worst case scenarios as only a limited number of players can interact in the same server partition at a given time. There have also been proposed several middleware systems for automatically distributing the game state among several participants. In [9], the authors present a middleware which allows game developers to create large, seamless virtual worlds and to migrate zones between servers. This approach does, however, not solve the challenge of many players that want to interact in a popular area. The research presented in [10] shows that proxy servers are needed to scale the number of players in the game, while the authors discuss the possibility of using grids as servers for MMOGs. Beskow et al. [3] have also been investigating partitioning and migration of game servers. Their approach uses core selection algorithms to locate the most optimal server. We have worked on how to reduce latency by modifying the TCP protocol to better support time-dependent applications [11]. However, the latency is not only determined by the network, but also the response time for the game servers. If the servers have a too large workload, the latency will suffer.

In [2], the authors are discussing the behavior and performance of multi-player game servers. They find that in the terms of benchmarking methodology, game servers are very different from other scientific workloads. Most of the sequentially implemented game servers can only support a limited numbers of players, and the bottlenecks in the servers are both game-related and network-related. The authors in [1] extend their work and use the computer game Quake to study the behavior of the game. When running on a server with up to eight processing cores the game suffers because of lock synchronization during request processing. High wait times due to workload imbalances at global synchronization points are also a challenge.

A large body of research exits on how to partition the server and scale the number of players by offloading to several servers. Modern game servers have also been parallelized to scale with more processors. However, a large amount of processing time is still wasted on lock synchronization, or the scaling is limited by partitioning requirements. In our game server design, we provide a complementary solution and try to eliminate the global synchronization points and locks, i.e., making the game server "embarrassingly parallel" which aims at increasing the number of concurrent users per machine.

VII. Conclusion

In this paper, we have shown that we can improve resource utilization by distributing load across multiple CPUs in a unified memory multi-processor system. This distribution is made possible by relaxing constraints to the ordering and atomicity of events. The system scales well, even in the case where all players must be aware of all other players and their actions. The thread pool system balances load well between the cores, and its queue-based nature means that no task is starved unless the entire system lacks resources. Message passing through the blocking queue allows objects to communicate intensively without blocking each other. Running our prototype game, we show that the 8-core server can handle twice as many clients before the response time becomes unacceptable.

VIII. Future work

From the research described in this paper, a series of further experiments present themselves. The relationship between linearly scaling load and quadratic load can be tweaked in our implementation. This could answer questions about which type of load scale better under multi-threaded implementations. Ideally, the approach presented here should be implemented in a full, complete massive multiplayer game. This should give results that are fully realistic, at least with respect to this specific game.

Another direction this work could be extended is to go beyond the single shared memory computer used and distribute the workload across clusters of computers. This could be achieved by implementing cross-server communication directly in the server code, or by using existing technology that makes cluster behave like shared memory machines.

Furthermore, all experiments described here were run with an update frequency of 10 Hz. This is good for many types of games, but different frequencies are relevant for different games. Investigating the effects of running with a higher or lower frequency of updates on server performance could yield interesting results.

If, during the implementation of a complex game, it is shown that some state changes must be atomic to keep the game state consistent, the message passing nature of this implementation means that we can use read-write-locks for any required blocking. If such cases are found, investigating how read-write-locking influence performance would be worthwhile.

REFERENCES

[1] A. Abdelkhalek and A. Bilas. Parallelization and performance of interactive multiplayer game servers. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 72, april 2004.

[2] A. Abdelkhalek, A. Bilas, and A. Moshovos. Behavior and performance of interactive multi-player game servers. *Cluster Computing*, 6:355–366, October 2003.

[3] P. B. Beskow, G. A. Erikstad, P. Halvorsen, and C. Griwodz. Evaluating ginnungagap: a middleware for migration of partial game-state utilizing core-selection for latency reduction. In *Proceedings of the 8th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 10:1–10:6, 2009.

[4] W. Cai, P. Xavier, S. J. Turner, and B.-S. Lee. A scalable architecture for supporting interactive games on the internet. In *Proceedings of the sixteenth workshop on Parallel and distributed simulation (PADS)*, pages 60–67, 2002.

[5] K.-T. Chen and C.-L. Lei. Network game design: hints and implications of player interaction. In *Proceedings of the workshop on Network and system support for games (NetGames)*, 2006.

[6] H. S. Chu. Building a simple yet powerful mmo game architecture. http://www.ibm.com/developerworks/architecture/library/ar-powerup1/, Sept. 2008.

[7] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, Nov. 2005.

[8] B. Drain. Eve evolved: Eve online's server model. http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/, Sept. 2008.

[9] F. Glinka, A. Ploß, J. Müller-lden, and S. Gorlatch. Rtf: a real-time framework for developing scalable multiplayer online games. In *Proceedings of the workshop on Network and system support for games (NetGames)*, pages 81–86, 2007.

[10] J. Müller and S. Gorlatch. Enhancing online computer games for grids. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 4671 of *Lecture Notes in Computer Science*, pages 80–95. Springer Berlin / Heidelberg, 2007.

[11] A. Petlund. *Improving latency for interactive, thin-stream applications over reliable transport*. Phd thesis, Simula Research Laboratory / University of Oslo, Unipub, Oslo, Norway, 2009.

[12] K. Raaen, H. Espeland, H. K. Stensland, A. Petlund, P. Halvorsen, and C. Griwodz. A demonstration of a lockless, relaxed atomicity state parallel game server (LEARS). In *Proceedings of the workshop on Network and system support for games (NetGames)*, pages 1–3, 2011.

[13] J. Waldo. Scaling in games and virtual worlds. *Commun. ACM*, 51:38–44, Aug. 2008.

[14] W. White, A. Demers, C. Koch, J. Gehrke, and R. Rajagopalan. Scaling games to epic proportions. In *Proceedings of the international conference on Management of data (SIGMOD)*, pages 31–42, 2007.

[15] W. White, B. Sowell, J. Gehrke, and A. Demers. Declarative processing for computer games. In *Proceedings of the ACM SIGGRAPH symposium on Video games (Sandbox)*, pages 23–30, 2008.