UNIVERSITY OF OSLO
Department of Informatics

The Influence of Latency on Short- and Long-Range Player Interactions in a Virtual Environment

Master thesis

Olga Bondarenko

November, 2012

The Influence of Latency on Short- and Long-Range Player Interactions in a Virtual Environment

Olga Bondarenko

November, 2012

Acknowledgements

I would like to express my gratitude to my supervisor, Carsten Griwodz, for guidance, valuable advice, useful ideas, inspiration, and patience, and to my second supervisor, Håkon Stensland, for encouragements and help with arranging the testing equipment.

Thanks to everyone who participated in the user studies, especially to my colleagues from USIT.

Finally, I would like to say thank you to my parents, my boyfriend and my friends, who always supported me.

Abstract

Multiplayer networked games, since the beginning of their history, have been continuously developing and gaining more and more popularity. Best-effort Internet has always been a real challenge for interactive online applications. Compared to earlier dial-up connections, home broadband has really become an improvement in network capacity, which also triggered the growth of the online games industry. However, due to higher latency sensitivity of some modern online multiplayer games, Internet latency can still be considered a bottleneck.

In this work we investigate the influence of latency on short- and long-range player interactions and determine the latency sensitivity of each category. We present findings from related literature and describe the process of the prototype implementation, considering the issues that were missing in related work.

To obtain the results, we perform user studies and evaluate their outcome. Finally, we conclude that short-range player interactions can tolerate considerably lower latency levels than long-range interactions.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The popularity of online multiplayer games today is hard to overlook. Online gaming, being not a particular gaming class, but rather a method used to connect players together over the Internet, has become a very common source of entertainment. Though single player online games are also common, the ability to compete and interact with other players over the Internet has always been more attractive than just confronting a computer. There are many reasons for that, and it is likely that every gamer can name their own motivations, but in general, interacting with characters controlled by other humans can make the game much more challenging and less predictable. Another big advantage of multiplayer games is social communication between players, which is missing in singleplayer games.

Multiplayer video games emerged already a few decades ago, though, the earliest ones supported only two players. Examples of such games are *Tennis for Two* (1958)[1], *Spacewar!* (1962)[2], *Pong* (1972)[3], and *Astro Race* (1973)[4]. Earliest networked multiplayer games, such as *Empire* (1973), were developed on the system called PLATO (Programmed Logic for Automated Teaching Operations), designed for computer-based education.[5] As we can see, networked games were introduced long before the Internet became available to the general public, and their development has been growing rapidly.

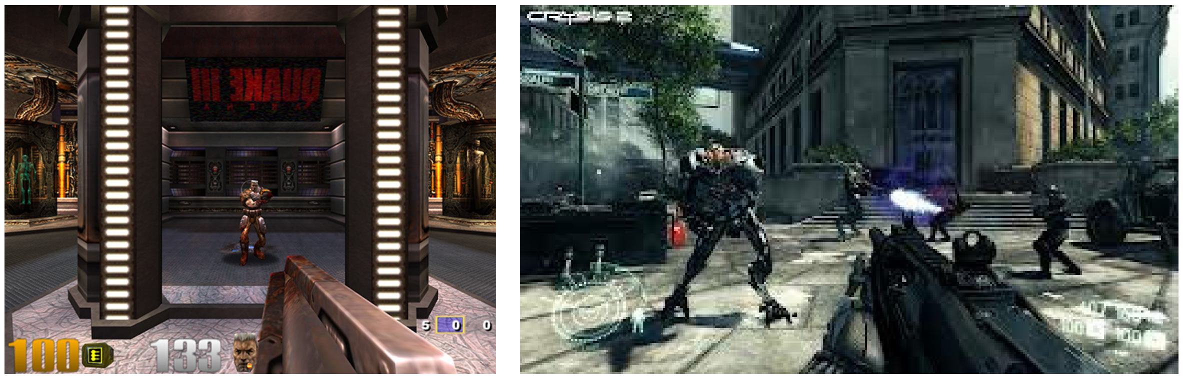
Still, mainstream availability of the Internet brought new potential to multiplayer gaming, making it possible to connect players all over the world. The number and variety

[1]http://scienceblogs.com/brookhaven/2010/12/14/resurrecting-one-of-the-worlds
[2]http://www.arcade-museum.com/game_detail.php?game_id=9074
[3]http://www.arcade-museum.com/game_detail.php?game_id=9074
[4]http://www.arcade-museum.com/game_detail.php?game_id=6949
[5]http://thinkofit.com/plato/dwplato.htm

of online multiplayer games currently available is very high. Generally, depending on the interactivity type, each of them belongs to some particular game model, though some genres cannot be easily classified. The most common models are **Avatar, First Person**, **Avatar, Third Person**, and **Omnipresent**. (Claypool and Claypool, 2006)

1.1.1 Avatar, First Person

Avatar, First Person is a model that allows players to experience game world through the eyes of a game character - avatar. Typically, an avatar is not visible at all, giving a player the feeling of "replacing" the avatar and being personally present in the game. The most common genre belonging to this model is First Person Shooter (FPS). In addition to the first person perspective, FPS games focus on shooting and combat. Most FPSs have a very fast game flow, enabling players to move around in the game world and compete with other players by aiming and shooting frequently. Though the avatar is typically not shown, arms and weapons might be displayed, as well as a status bar showing health condition and ammunition. Such games often require quick reaction and focus from the player. Due to advanced 3D graphics with high requirements to hardware, FPSs only started to spread in 1990s. Some well-known examples of FPSs are *Wolfenstein3D* (1992), *Doom* (1993), and *Quake III* (1999). Recent FPSs released in 2011 are *Bulletstorm* and *Crysis 2*.



(a) Quake III (b) Crysis 2

Figure 1.1: First Person Avatar games

1.1.2 Avatar, Third Person

Avatar games with a third person perspective give players the ability to see the avatar on the screen and follow it in the game world. One of the most popular third-person avatar genres is Role Playing Games (RPG). Inspired by an earlier tradition on role-playing, the genre has developed from being a simple hobby to commercially important part of the gaming industry. In RPGs, users typically interact by playing some particular roles of the characters in some fictional setting. Role-playing is, in other words, collaborative story-telling, usually performed by certain actions, skillful thinking combined with strategic acting, or character development. Success of a player is then determined by the game's rules and guidelines. (Harrigan and Wardrip-Fruin, 2010) The first marketed RPG example is *Dungeons & Dragons* (1974), while recent examples are *Dragon Age II* (2011), illustrated on Figure 1.2(a), and *Dogfight* (2011).

Some third person avatar genres can be similar to those, belonging to the first person avatar model. For example, Third Person Shooter (TPS) is closely related to FPS, mostly differing in the view perspective. Similarly, other genres are present in both first and third person avatar models, such as sports games and racing simulators. However, racing games are mostly played in a third person avatar view. The genre usually involves controlling a vehicle, typically, a racing car, though any type of vehicle can be used. Being a subgenre of simulation video games, racing games can incorporate anything between simple car races and hardcore simulations; one of the early examples is *Space Race* (1973), while a recent one is *F1 2011* (2011), shown on Figure 1.2(b).

(a) Dragon Age II (b) F1 2011

Figure 1.2: Third Person Avatar games

1.1.3 Omnipresent

Omnipresent game model is based on player's ability to be present everywhere. As opposed to first or third person avatar, varying perspective of omnipresent games often allows using several views. Typically, a player is able to see the virtual world from the bird's eye view, zoom in to see it through the eyes of a character, or zoom in anywhere to control the desired details. (Claypool and Claypool, 2006) Typical genres belonging to this model are Real Time Strategy (RTS) and again, different types of simulation video games.

RTS is one of the four subtypes of the strategy genre, where the main idea is achieving success by using skillful thinking and strategic planning. Subtypes depend on whether the game is real-time or turn-based, and whether it targets strategy or tactics accordingly. (Rollings and Adams, 2003)

RTS games, as one can see from the title, focus on real-time setting and strategy mode. As most of the strategy games, RTS are typically war games, offering participants such features as resource gathering, base construction, technological growth, and control of units. Units are usually moved to a different location, manipulated to control certain map areas or destroy enemy's assets. It is usually possible to add new units during the game by spending the resources gathered previously. Resource gathering is typically achieved by controlling certain map points and/or obtaining a particular type of units or structures, dedicated to this purpose.[6] Popular recent RTS games include *Warcraft III* (2002) and *Starcraft II* (2010), illustrated on Figure 1.3.

The variety of simulation games is huge. Generally, they attempt to simulate some types of real-life activities. Such activities can include building and maintaining an entire city, as in *SimCity*, released first in 1989 and last in 2013. Omnipresent simulation games are often used not only for entertainment purposes, but also as educational and training applications.

(a) Warcraft III

(b) Starcraft II

Figure 1.3: Omnipresent games

1.1.4 Massively Multiplayer Online Games

A Massively Multiplayer Online Game (MMOG), like an online game in general, is not a particular model or genre. Any online video game, capable of supporting hundreds or even thousands of simultaneous players, is considered a MMOG. There are various genres of MMOGs, as well as their mixtures. Popular MMOGs include such genres as Massively Multiplayer Online Role Playing Game (MMORPG), Massively Multiplayer Online First Person Shooter (MMOFPS), and Massively Multiplayer Online Real-Time Strategy (MMORTS). The games that were placed on the top of the most popular MMOGs in 2012 list are *Guild Wars 2*, *Star Wars: The Old Republic*, and *World of Warcraft* [7], depicted on Figure 1.4.

[6]http://pc.ign.com/articles/700/700747p1.html
[7]http://www.mmorpg.com/showFeature.cfm/loadFeature/5524

(a) Guild Wars 2 (b) Star Wars: The Old Republic (c) World of Warcraft

Figure 1.4: MMOGs

1.1.5 Network Latency Challenge

A networked group multimedia application is typically influenced by the network's Quality of Service (QoS). The main parameters of QoS, in regard to networked applications, include throughput, transit delay, delay jitter, error rate, and degree of reliability. Depending on the specific application type and properties, each of these factors has different influence. For example, a video conference application would be negatively influenced by high jitter, but tolerate high loss level, while a shared whiteboard would be highly sensitive to loss, but tolerable to low bandwidth. (Mathy et al., 1999; Henderson and Bhatti, 2003)

A networked multiplayer game involves exchanging messages, usually between clients and the server. Typically, such messages contain data about players' states and ideally, need to be delivered fast. However, encoding data into a message, its delivery, and processing by the receiver take certain time, with greater part of it spent on delivery. Therefore, multiplayer games played over the Internet can be significantly influenced by transit delay, often referred to as **network latency**.

In a packet-switched network, network latency is either a one-way time delay measured from the moment a packet is transmitted from the source to the moment it is received by the destination, or a round-trip (RTT) delay - a sum of one-way latency from the source to the destination and back. Round-trip delay is used more often and is typically estimated by using a **ping** service. Ping does not process packets and only sends a response back after receiving a packet, which makes it a convenient way of calculating latency. (Comer, 2000)

Best-effort Internet has always been a real challenge for multiplayer online games. Compared to earlier dial-up connections, home broadband has really become an improvement in network capacity, which also triggered the growth of the online games industry. However, due to higher latency sensitivity of some modern online multiplayer games, Internet latency can still be considered a bottleneck. Compared to LAN latency of less than 10 ms, Internet latencies can be relatively high. Dial-up modems used to add hundreds of milliseconds of latency, while in broadband access networks (cable or asymmetric subscriber lines), it is reduced to tens of milliseconds. However, cable modem latency usually varies and can be even higher than 100 ms. Geographical location is another factor that influences network latency. Inside the continent, latencies are typically within 50 ms, getting higher across continents. Overall latency on the Internet can vary from hundreds of milliseconds to even more than a second. (Claypool and Claypool, 2006)

A number of studies have determined that network latency can influence player performance. The overall effect depends on the delay rate and latency sensitivity of a particular game model or genre. Chapter 2 presents more details about the results of such studies.

Knowing latency sensitivity limits can be useful for:

- *game designers*, in designing games with regard to it and applying necessary techniques to hide it;
- *network designers*, in creating proper infrastructures and offering sufficient quality of Internet connections;
- *Internet-based game providers*, in planning the locations of game servers;
- *players*, in making informed Internet provider choice and being aware of their performance chances in playing a particular game. (Claypool and Claypool, 2006; Armitage, 2003)

1.2 Problem Definition / Statement

Some research has been done to determine the influence of latency on player performance in different game genres, focusing typically on the model properties and player actions that are common for it. However, new games emerge, often belonging to an existing game model, but incorporating various details that can significantly influence latency sensitivity. To the best of our knowledge, no research has been done to determine latency sensitivity of multiplayer online games that use human models as game characters and implement player interactions, where distance between avatars plays an important role in successfully fulfilling an action.

Human characters are already being used in many video games and applications. We believe that human avatar perspective can become even more popular in future. The idea of simulating real world has often been a goal of designing certain tools or products. Usage of human avatars significantly contributes to the real-world experience by allowing users to undertake actions that are possible in real life. It can be applied not only in video games, but also used for many other non-entertainment purposes, such as video conferencing applications, where the opponents are represented by avatars, or any other types of educational or training simulators.

Natural human behavior during interactions typically involves immediate reaction and response. Consider a simple example where two persons are having a conversation in real life. Each of them is either listening, talking, articulating, moving, maintaining eye contact, or simply being present and visible to the opponent. As a result, we get a complex interaction that has a certain flow and generally, excludes any "gaps", where any or both opponents are totally inactive. Situations when an opponent suddenly stops talking without finishing a sentence, or walks away unexpectedly, are not perceived as natural. A person cannot suddenly disappear from the view, move over great distances in a blink of an eye, or do anything that is physically impossible. Thus, a human interaction typically involves certain constraints and expectations, in addition to the requirement of immediate response.

As described in Chapter 2, the literature states that in some typical game genres, such as FPS, different interaction categories are not equally sensitive to network latency. For instance, it was observed that precision shooting was more sensitive to network delay than other player interactions, due to the importance of knowing correct players' positions at the time of the shooting event. However, it was not stated whether the distance between players at the moment of shooting event was of any significance. Therefore, the purpose of this thesis is to investigate how latency affects user performance and gameplay experience in a virtual environment with human avatar perspective, featuring short- and long-range player actions that simulate real-life human interactions.

Simulating full physical presence of a human in interactive applications is by far not an easy task. Actions assigned to human characters in video games are usually simpler, but may still require fast response, due to the nature of human behavior that is being simulated. Therefore, instead of simulating full physical presence, our goal is to define main categories of player interactions and study the effect of latency on the outcome of each interaction.

1.3 Limitations

This work requires further analysis in certain areas. One can suggest that high latencies are often caused by long network paths, which means that such paths could have greater jitter. We do not take the presence of network jitter into account. Therefore, we do not present data indicating whether it can influence player performance or gameplay experience. Similarly, measuring the influence of packet loss is needed, which is neither considered in this work.

1.4 Research Method

The goal of this work is to determine how short- and long-range human interactions, used as avatar actions, are affected by increased levels of latency. To achieve the desired outcome, we first study the literature that focuses on the problems related to network latency in the context of multiplayer online games. Further, we design an interaction model based on findings from relevant literature, also considering issues that were not addressed in related work, and implement a prototype. Eventually, we conduct user studies by performing the experiments in the test environment and evaluate the results. The metrics selected for analysis is mainly based on the objective score achieved by players in the end of the game, but also includes observations of players' behavior and reaction.

1.5 Main Contributions

In this work we have presented findings from literature that studied the effect of network delay on player performance and introduced the concept of player interaction range. Since the issue of interaction range was not addressed in the related work, we have built a prototype, allowing the investigation of short- and long-range player interactions, and presented the details of its design and implementation.

The results of our experiments proved that short-range player interactions are considerably more sensitive to network latency than long-range interactions.

1.6 Outline

The structure of this work is organized as follows.

- In Chapter 2, we discuss literature that focuses on the influence of network delay on different game genres and point out some important factors that were not considered.

- Chapter 3 presents the details of designing and implementing a prototype, incorporating the issues that were not addressed in the related work, as well as the discussion of different implementation alternatives and arguments for the solution choices.

- In Chapter 4, we present our hypothesis and describe the experiments, conducted to verify it. Further, we discuss the results and explain the metrics.

- Chapter 5 provides a short summary of the thesis, lists main contributions, and gives an outlook of remaining work.

Chapter 2

Related Work

In this chapter, we analyze related work that focuses on determining how different game genres react to the impaired network QoS, latency in particular. The chapter is organized as follows. In the first three sections, we describe related work on First Person Avatar, Third Person Avatar, and Omnipresent game models, evaluating latency sensitivity of certain player interactions and players' perceptions of the game quality under varying network conditions. Further, we discuss and summarize the results.

2.1 Avatar, First Person

The most common genre belonging to the first-person avatar model is FPS. Since FPS games typically require quick reaction to the game events from the user, one can assume that the changes in the user state need to be delivered to other players as soon as possible. For this reason, FPS players often believe that their performance can be influenced by the network conditions. This assumption is fully justified. Moreover, several studies that focused on determining the impact of varying network conditions on the final outcome of FPS games prove that.

Armitage (2003), Henderson (2001), and Henderson and Bhatti (2003) used publicly accessible game servers to determine how increased delay affected players' decisions regarding joining a particular game server or leaving the game due to dissatisfaction. Generally, the authors observed the behavior of players connecting to the servers at different levels of latency. It was hypothesized that increased delay could affect the users in two ways - they would give up to join a game or leave the game after observing higher delay. At the same time, it was assumed and that players returning to play again were mostly satisfied with their experience. The studies were based on popular FPSs, namely, Quake III and Half Life. Armitage (2003) and Henderson (2001) measured each player's ping time, while Henderson and Bhatti (2003) also intentionally introduced some extra delay. In addition to that, Armitage (2003) and Henderson and Bhatti (2003) measured the number of times a player was killed or killed an opponent and investigated how the kills/deaths rate was influenced by impaired network conditions.

The results of all the three studies were similar. Armitage (2003) figured out that that Quake III players were unlikely to connect to a server with latency above 150-180 ms, while the average frag (kill) rate per minute started to decrease already at latency over 50 ms. For Half-Life, Henderson (2001) concluded that delay played an important role in player's decision to join and stay on the server, causing users to give up and look for a different server in case of delay over 225-250 ms. Similarly, Henderson and Bhatti (2003) observed that additional delay of over 300 ms, introduced at the Half-Life server, started to noticeably reduce the average number of players joining the game, while the number of players leaving the game increased; quantitative results indicated that as delay increased from 25 ms to 250 ms, the average number of kills per minute dropped from 1.456 to 0.6233, and the average number of player's deaths per minute increased from 0.6042 to 1.430.

We can clearly see that players' performance measured by kills/deaths rate was best at the lowest latency levels. However, even though the authors analyzed a great amount of data obtained from real-life users, none of them considered players' skills and experience, which could also have affected their performance, as well as differences in hardware used for playing. Also, one can suggest that there were other reasons, apart from dissatisfaction with the game quality, that could have affected players' joining and leaving decisions. Nevertheless, no other factors were examined, since it was impossible to obtain additional information about the users. Neither did these studies attempt to categorize player interactions and analyze how each interaction was affected by varying network environment.

In contrast, the works by Beigbeder et al. (2004), Dick et al. (2005), Wattimena et al. (2006), Quax et al. (2004), and Zander and Armitage (2004) are based on the controlled environment approach, analyzing player performance and perception of the game by investigating some factors that were impossible to consider with the public-server based approach, such as player interaction categories and user skills. The FPSs used for analysis include Unreal Tournament 2003/2004, Quake III/IV, Counter Strike, and Halo. Beigbeder et al. (2004) examined delay and packet loss tolerance of different player interactions, namely, movement, shooting, and shooting while moving. For measurements, the authors used Mean Opinion Score (MOS), a quite common measurement of player's subjective perception of the game, where the game environment is given a score value between 1 (unacceptable) and 5 (perfect) by the players themselves, and objective quantitative performance measurements, based on players' kills/deaths rate. All the authors, except Quax et al. (2004), took players' skills into account.

The overall results were similar to those, derived from public server based works. Beigbeder et al. (2004) state that simple movement (walking or running in a straight line) was not affected by delayed or lost packets, while complex movement (jumping, spinning, and navigating through obstacles) required the player the use more time to complete the experiment at latency over 300 ms. Precision shooting appeared to be generally tolerable to packet loss, but rather sensitive to latency - delays over 75 ms resulted in a steady hit accuracy decrease. At 100 ms of delay, the number of kills was decreased, while the number of deaths increased by 35%. When shooting was combined with movement, the authors observed a downward linear trend in the kills rate and an upward linear trend in deaths rate at latency over 100 ms, while the total degradation of performance was approximately 30% at 200 ms of delay.

The results presented by Dick et al. (2005) indicated that on average, players experienced Counter Strike as still playable at the delay of 500 ms, while Unreal Tournament 2004 was perceived as annoying environment at delays over 150 ms; variance in jitter seemed to have no significant effect on players' perception of the games; objective score results for Counter Strike were undetermined, while in Unreal Tournament 2004, the Game Outcome Score (GOS) dropped significantly (by over 50%) at delays over 150 ms; jitter seemed to have no significant effect on GOS score in Unreal Tournament 2004, whereas results for Counter Strike were again undetermined.

Wattimena et al. (2006) concluded that delay and jitter negatively affected gaming experience and objective score results of only expert and super-expert players participating in the experiments, while Quax et al. (2004) came to the same conclusion without considering the differences in players' skills. Zander and Armitage (2004) discovered that latency above 200 ms caused degradation of player performance, mentioning that players with higher skills were affected more than bad players; the subjective ratings indicated that the perceived game quality started to decrease at latencies over 200-300 ms, resulting in 20-40% of players wanting to leave the game, while quality drops below average occurred in the range of 300-400 ms.

The works discussed in this chapter took different QoS factors into account. As we can see from the general outcome, latency was the most significant factor that led to players' performance degradation and lower subjective ratings of the game quality. In addition to the research focused directly on FPSs, the study by Claypool and Claypool (2006) compared the effect of network latency on three different game models. The overall results proved that games belonging to the first person avatar model were most demanding to QoS, being particularly sensitive to network latency. However, none of the studies considered which factors in particular contributed to the latency sensitivity of FPSs, except the high pace of this genre's gameplay and the necessity to react quickly. Therefore, it was not determined whether the distance between players had any significance.

2.2 Avatar, Third Person

One of the most prominent genres representing the third person avatar model is MMORPGs (massively multiplayer online role-playing game). The great popularity of MMORPGs is demonstrated by constantly high number of active users, meaning that gamers enjoy playing it. Since MMORPGs connect many players throughout the world, it is a reason to suppose that the genre is not significantly affected by Internet latencies. Evidence for that is present in the works by Fritsch et al. (2005) and Chen et al. (2009).

The paper by Fritsch et al. (2005), based on controlled environment approach, focused on Everquest II, an example of second generation MMORPGs. Two most important interactions present in the game were pointed out - movement with combat and group combat. The results of the tests indicated that the game was no longer playable at latency over 1250 ms. However, combat was still rather accurate even at 500-1000 ms and the player abilities were used coordinated at latency of up to 1000 ms, which points out that the game tolerated latencies of up to one second. Nevertheless, this fact does not prove that any third person avatar game has equivalent requirements to the network QoS. One of the reasons for that is the way player performance was evaluated. Everquest II does not define any final goal, while there are some aspects that can make a player "better", so, the measurement of player's success can be based only on certain player characteristics. Interestingly, maintaining constant position consistency seems to have insignificant influence on the game outcome. Though the authors observed that maintaining consistency of players' states was still necessary, delivering updates within a second was acceptable due to the compensating mechanisms used in the game. The interaction model that we implement employs third person avatar perspective as well. However, we believe that human interactions are generally dependent on players' positions consistency and, as a result, have stricter requirements to network QoS.

In contrast to MMORPGs, in racing video games, a genre that generally uses third person avatar perspective, maintaining position consistency considerably affects the outcome of the game; therefore, the genre is more sensitive to network delay. Pantel and Wolf (2002a) evaluated two racing games - Re-Volt and Need-for-Speed, in two player mode, using 2 PCs. The results of the tests showed that that delay handling in these two games was not sufficient already at 100-200 ms of latency, mostly resulting in the local car leading on each computer respectively or causing a significant discrimination of the car driving behind during collisions. In addition, the authors conducted some experiments with the car racing simulator developed for investigating the same problem. Three players with different skill levels participated in the tests, and the time spent to complete a round was used as a performance measurement. The results have shown certain differences in influence of higher latency on player performance, depending on player's expertise. The beginner faced no degradation in performance at delays of up to 50 ms, but also seemed to tolerate well latencies of up to 200 ms, most likely due to not being experienced enough to obtain benefits from a fast and responsive system and generally, driving rather slowly. The medium-level driver with a rough driving style reacted faster to changes in driving direction and could therefore not tolerate high delays, showing worse results at delays over 50 ms. The experienced driver performed well until the delay reached 150 ms, but starting from that point, an almost exponential performance degradation was observed. In general, the results from all the three players indicate that only delays below 50 ms did not cause significant changes in player performance. Since racing simulators typically use similar interaction models, we can assume that the 50 ms latency limit applies to the genre on the whole.

It is interesting to observe that related work on third person avatar games presents completely opposite results in regard to latency sensitivity of different game genres. Racing simulators, where players' score depends on how well the position consistency is maintained, are characterized by low threshold of tolerated latency, while MMORPGs, where position consistency is less important, can in turn tolerate higher delays. However, no third person avatar games with short-range interactions between players were evaluated. Mortal Kombat, that has never been released as online game, is a good example. Being a popular third person avatar fighting game since 1990s, it is entirely based on short-range player interactions. The game was developed initially for arcade machines and eventually for home consoles. In April 2011, it became available for PlayStation 3, and for PlayStation Vita in 2012. During a media event, where Mortal Kombat was advertised, its co-founder, NetherRealm Studios creative director Ed Boon, was asked whether the game supported online play. His reply was:

> "It's just the Wi-Fi. Fighting games are a really twitchy, latency-sensitive experience. We don't want to expose people to the idiosyncrasies of wireless carriers and lead them to have a bad experience. So you can get on Wi-Fi or you can play against someone locally. It's great on Wi-Fi." [1]

Such answer points out that Mortal Kombat's developers were aware of the latency sensitivity problem and considered the game's delay threshold to be much lower than typical Internet latencies. Thus, the tolerable latency limit for fighting games that are similar to Mortal Kombat is unclear.

2.3 Omnipresent

Omnipresent game model is typically represented by the RTS genre. Claypool (2005) has studied how latency influenced game outcome in such RTSs as Blizzard's Warcraft III, Microsoft and Ensemble Studios' Age of Mythology, and Electronic Arts' Command and Conquer: Generals. To determine the effect of latency on user performance, typical user interactions were divided into three categories - building, exploration, and

[1] http://www.theglobeandmail.com/technology/gaming/controller-freak/qa-10-minutes-with-mortal-kombats-co-creator/article547669/

combat. The overall results of the tests, performed on all the three games, showed that latency in the range of hundreds of milliseconds to several seconds could still be tolerated. While such outcome can seem rather surprising, it can be explained by nature of the RTS genre, where strategy is much more significant than interactive aspects.

2.4 Discussion

Table 2.1 summarizes the thresholds of latencies tolerable by games belonging to each of the three models, as presented in related work.

Table 2.1: Tolerable latency thresholds for different game models

Model	**Game**	**Distance significance**	**Latency threshhold (ms)**
Avatar, First Person	Quake III/IV	uncertain	150-200
	Half-Life	uncertain	225-300
	Unreal Tournament 2003/2004	uncertain	100-150
Avatar, Third Person	Everquest II	no	1000
	Racing simulators	yes	50
	Mortal Kombat	yes	N/A
Omnipresent	Warcraft, other RTSs	no	1000

The third column in the table indicates whether the distance between players in the game world plays an important role during interactions. For example, if a typical player interaction is only possible to fulfill at certain (mostly very short) distance between players, like injuring an opponent in Mortal Kombat, or, if winning is dependent on player's location, as in Re-Volt or Need for Speed, distance is important, whereas in games, where player performance is mostly influenced by other factors, distance between players is not important. Since the distance factor was not taken into account in evaluating FPSs, as mentioned in Section 2.1, we conclude that, in such case, it is uncertain whether it is significant.

Looking at the results of commercial games evaluation in Table 2.1, we can observe that car racing simulators (Re-Volt, Need for Speed) have low latency threshold (50 ms). We suppose it is caused by the short-range interactions influencing players' score. Nonetheless, such correlation was not taken into account in any of the works that analyzed human avatar based games. Moreover, Mortal Combat, based on short-range player interactions, does not even support online play functionality due to high latency sensitivity, as stated in Section 2.1. Therefore, we believe that further analysis is needed to determine how short- and long-range interactions in human avatar based games are affected by varying network delay.

2.5 Summary

In this chapter, we discussed useful findings in the literature that studied the effect of network latency on different game genres. We pointed out that though related work differentiated between various types of player interactions and examined latency sensitivity of each type, the difference between short- and long-range player interactions was not considered. Therefore, in the next chapter we describe design and implementation of a prototype that takes the missing issue into account.

Chapter 3

Game Model Design and Implementation

Since we figured out that related work investigating the influence of latency on player performance did not consider such issues as human interactions and distance range, we need to build a prototype that takes these factors into account. This chapter is based on the details of designing and implementing a prototype of a player interaction model and is organized as follows. In the first two sections, we discuss the alternatives for designing a 3D model of a human avatar, animating it and integrating it into game development environment. Thereafter, we discuss the categories of player interactions. Further, we describe the details of physics simulation and collision detection implementation. The third section focuses on client-server protocol, interpolation, client-side prediction, and latency compensating techniques.

3.1 3D Model of a Human Character

The development of graphic techniques applied in video games started from rather simple strategies, such as using text characters to depict objects, actions or any other aspects of a game world. Advances in hardware and power of central and graphics processing units have continuously been enhancing graphic techniques, making game world look more real.

Graphic techniques used in many modern video games are based on a three-dimensional representation of geometrical data, often in a form of Cartesian coordinates. Entities in a 3D game world consist of a collection of points in the coordinate system and are connected by lines, surfaces, and other geometrical objects. Such entities are usually referred to as 3D models.

There are two main categories of 3D models representation - **solid** and **shell/boundary**. Solid models are generally more complicated, since they define the volume of an object. Shell models do not represent volume, but only the surface instead. Generally, most of the 3D models used in computer games are shell models. The main reason for that is that computer games mainly make use of object's visual representation and not its volume. Also, shell models are easier to develop.

There are different ways of developing 3D models; one of them is **polygonal modeling**. It is widely used for making 3D objects, such as avatars in computer games. The approach is based on using polygons for representing object's surfaces. An object represented by polygons is usually referred to as **polygonal mesh**. Polygonal mesh includes vertices, edges, faces, polygons, and surfaces, that determine object's shape.

Let us consider a simple polygonal mesh, visually representing a cube, as shown on Figure 3.1. [1]

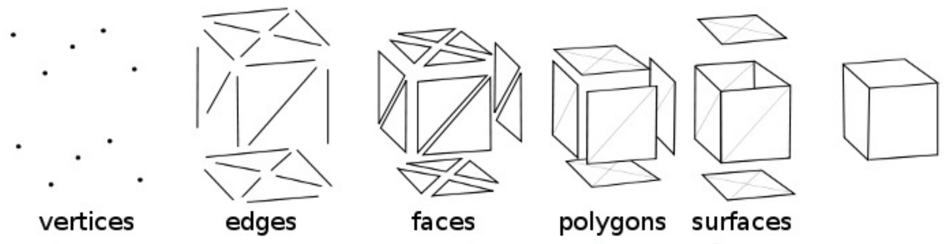


Figure 3.1: Elements of polygonal mesh

In this example, every corner of the cube is a **vertex**, and a line connecting two vertices is an **edge**. Not only angle points are vertices. A vertex is also a point where two or more lines intersect, for instance, when a line separates different colors or textures. A closed set of three or more edges forms a **face** or a **polygon**. If multiple-sided faces are supported, then there is no need to differentiate between faces and polygons, but if rendering hardware supports only three or four sided faces, then polygons are broken into faces. A **surface** is an optional element of polygonal mesh. Surfaces are often referred to as smoothing groups and are used to group smooth regions.

Polygonal mesh data can be stored in different formats. Some of the popular formats are:

- *.blend* - Blender file format
- *.3ds* - 3D Studio Max file format
- *.fbx* - Autodesk file format

[1]The figure is inspired by Wikimedia Commons file http://en.wikipedia.org/wiki/File:Mesh_overview.svg

- *.dae* - Digital Asset Exchange (COLLADA)
- *.obj* - Wavefront object
- *.mesh* - OGRE binary mesh format

One can use 3D modeling software for creating a 3D mesh and storing it in some file format. There are many examples of 3D modeling software, but in terms of use, software tools can generally be divided into two main categories - proprietary and free and/or open source. Popular commercial examples include Autodesk 3D Studio Max and Autodesk Maya, which are only available for Windows platform. Examples of free and open source 3D modeling software are Blender and Art of Illusion. Both support Windows, Linux, and Mac OS. When it comes to software choice, usually, the decision depends significantly on one's requirements and expectations.

Blender is an example of quite well-known free and open-source tool, providing almost equivalent range of features to those, available in high-range commercial software. It is widely used for 3D modeling, creating animations, interactive applications, and computer games, including also some other features. Blender was first developed in 1989 as an in-house project by a company called Not a Number Technologies (NaN). In 2002 NaN went bankrupt and the project was commercialized. Shortly after that, when enough funds were collected, Blender source code was released under the terms of GNU General Public License.

It is important to notice that Blender interface and some of its features, such as animation system, were significantly changed in version 2.5. For this reason, some projects created in Blender versions before 2.5, in particular those containing animated mesh, are not anymore supported by the most recent versions, starting from 2.5.

3D modeling in Blender can either be done from scratch or by making use of available 3D models. 3D mesh obtained as a final result can be animated. There are two main techniques used to animate a character consisting of a 3D mesh - **per-vertex animation**, also called **morph target animation**, and **skeletal animation**. In a per-vertex animation, the mesh is deformed by changing the positions of its vertices manually and interpolating between them. In the context of 3D models used as avatars for computer games, this type of animation is most commonly used for facial or cloth animation. For animating avatar's actions, such as walking, jumping or any other movements, skeletal animation is typically used.

Skeletal animation implies adding a **skeleton** to the mesh. A **skeleton**, (also referred to as **armature**), is a set of **bones**. A bone is simply a group of vertices, while positions of those vertices define its size, location, and orientation. In a skeleton, bones are usually represented as a parent-child hierarchy. A bone that does not have a parent is usually referred to as **root bone**. Each vertex on the mesh can be weighed to one or more skeleton bones, creating a parent-child relationship between skeleton and mesh. This process is referred to as **rigging**, and a 3D model consisting of mesh and skeleton, where the skeleton is parented to the mesh, is called a **rig**. A rigged character can be animated by manipulating the positions and orientations of the skeleton bones, which in turn transform the positions of mesh vertices weighed to them. There are two main techniques to compute bone positions in skeletal animation - **forward kinematics** (FK) and **inverse kinematics** (IK).

In FK, rotating or translating a bone in a parent-child hierarchy affects all the child bones (if any) respectively. Total transformation of a bone is then defined by its own transformation multiplied by total parent transformation. Translating or rotating a bone in a skeleton thereby only affects the manipulated bone and all its children, but none of the parent bones. FK is usually the default technique applied to skeletal animation.

IK implies using **end effectors** to control a rig. In skeletal animation, an end effector is a control point, usually assigned to a bone. Any bone can be assigned an end effector, but most common end effectors for a human character are bones representing feet, hands, elbows, and knees. Choosing target positions for end effectors creates a problem of computing the corresponding positions of all the other bones in the skeleton. The problem can be solved in different ways, and most of the solutions usually come from robotics applications. (Li et al., 2009)

Blender has an automatic IK solver that uses any selected bone as an end effector. However, that automatic IK solver is rather inefficient tool for creating animations and is mostly used for quick demonstrations. The most common way of applying IK in Blender is adding extra bones used as end effectors and assigning IK constraints to them. An IK constraint defines which bones in the chain are affected when end effector is manipulated. [2]

Some existing 3D models are already rigged, and sometimes, they include a set of animations. Such rigs often differ in complexity of mesh and armature structure. More complex mesh structure makes its visual representation more precise and smooth, while advances in skeleton structure simplify the character animation process. Whether one should choose more or less advanced structure of a 3D model usually depends on desired functionality, type of 3D modeling software, and rendering techniques used.

Considering 3D models of human characters, there is a relatively good example called *ManCandy* - a rig created with Blender. It contains polygonal mesh and rather advanced skeleton structure, as shown on Figure 3.2. There were three official versions of *ManCandy* (1.0, 2.0 and 2.1), and all of them were developed by Bassam Kurdali, the director of *Elephant's Dream*, the first movie made with Blender. Those versions were fully functional in Blender 2.45. Due to significant changes in Blender since version 2.5, some of the *ManCandy's* features were not supported anymore. Shortly after Blender 2.5 release, *ManCandy* was updated by a Blender artist Wayne Dixon and became com-

[2]Blender user manual describes usage of IK in more detail: http://wiki.blender.org/index.php/Doc:2.6/Manual/Rigging/Posing/Inverse_Kinematics

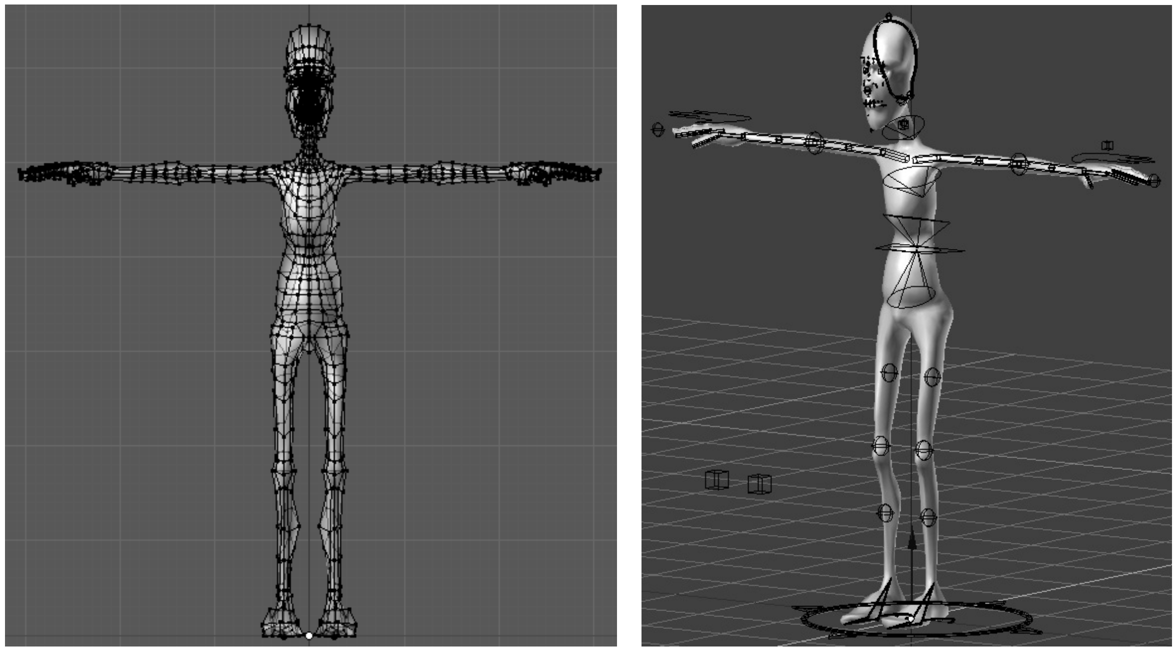

(a) Mesh structure (b) Main skeleton bones

Figure 3.2: *ManCandy*

patible with Blender 2.5.

ManCandy is a complicated rig that requires certain knowledge of Blender features and experience in 3D modeling to understand its components and make changes to the rig itself. It is worth noticing that the character itself was created mostly for making animated videos.

Animating a 3D character in Blender is usually done by using **modifiers** - automatic operations that change the way an object is rendered, but not the actual geometry of the mesh. There are fours groups of modifiers in Blender - modify, generate, deform and simulate. Modifiers from modify group are used to transform mesh without directly affecting its shape. Generate group modifiers are tools that change the actual geometry of the mesh by either modifying its appearance or adding new geometry to the object. The function of deform group modifiers is changing object's shape. Modifiers from this group are widely used in creating animations, so, it is worth paying more attention to its members:

- *Armature* - adds skeletal animation to an object.
- *Cast* - allows to shift the shape of a mesh, surface or lattice to a sphere, cylinder or cuboid.
- *Curve* - enables bending an object using a curve trajectory.
- *Displace* - uses a texture to deform an object.

- *Hook* - adds a hook to vertices to manipulate them externally.
- *Lattice* - deforms selected object using a lattice object.
- *Mesh Deform* - deforms a mesh object by changing the shape of another mesh object.
- *Shrinkwrap* - allows shrinking or wrapping an object around the surface of another mesh object.
- *Simple Deform* - used to apply advanced deformations to an object.
- *Smooth* - smoothens mesh geometry.
- *Warp* - stretches a mesh object between two specified points.
- *Wave* - deforms an object to shape waves that can be animated.

Modifiers from simulate group are used to activate simulations. One can add several modifiers to an object and form a stack of modifiers, and each modifier can be applied to make the changes permanent. Blender user manual describes modifiers in more detail. [3]

The *Mancandy* rig combines armature modifier with some other modifiers from the deform group (curves and lattices), that are supported by Blender, but not always supported by other software, such as game engines. Using the model as a game character can require some modifications and optimizations, depending on the game engine one is going to use. Blender Game Engine (BGE) allows using modifiers, but they can create a large computational overhead during runtime. Generally, it is recommended to apply or disable any modifiers, excluding armature modifier, before using a Blender model in a game engine.

Optimizing *Mancandy* to be used in BGE can be done by disabling or applying non-armature modifiers. In case of using it in a different game engine, .blend format needs to be converted to the format supported by the chosen software. However, exporting the rig to a different format is a complicated task.

[3]More information can be found at Blender Manual webpage about modifiers: http://wiki.blender.org/index.php/Doc:2.6/Manual/Modifiers

3.2 Game Engine

3.2.1 Game Development Tools

When first video games emerged, the capabilities and processing power of computers were rather low. During that time, a game was usually developed as a single entity, containing all necessary components for communicating with an operating system or hardware directly. Due to ever-increasing processing power of modern computers and raising demands to what a computer game should be, different tools were designed to simplify the process of game development, such as **game engines**.

A game engine provides a set of game development tools and reusable software components for faster and simpler game creation. Apart from graphics rendering functionality, game engines can contain a wide range of components that are commonly used in games, such as physics engine, sound, animation tools, scripting, networking, threading, and others. Examples of such game engines are Panda3D, Unity, Adventure Game Studio, and BGE.

Software providing graphics rendering is often referred to as a **rendering engine**, or **3D engine**, if it is designed to render 3D graphics in particular. Examples of popular 3D engines are OGRE, Irrlicht, Genesis3D, and Horde3D. A 3D engine can be combined with other libraries to build a game engine functionality. (Zerbst, 2004)

For the purpose of creating a test game model, the factors that significantly influence the choice of development tools are ease of use, full control over the source code, and, particularly in our case, the ability to use human characters.

Choosing BGE would have an advantage of using 3D models created in Blender directly, without the need to export them. Also, it includes an integrated physics engine, sound, and input processing libraries. However, there are some disadvantages. The engine provides a graphical "logic bricks" interface, consisting of "sensors", "controllers", and "actuators". A sensor is added to a game object and associated with some event, like keyboard input or collision. A controller processes the input from one or several sensors and triggers one or more actuators that provide responses. A Python script can be added to a controller to handle sensor input instead of triggering an actuator, but not all Python modules are fully supported. For example, threading module can only be used effectively if the threads finish up before the script itself.[4] The main purpose of using the "logic bricks" concept was to make the game engine intuitive and easy to use for artists, in particular, Blender artists. Creating a game in BGE does not give a possibility to see entire source code, which makes it difficult to extend, debug or maintain the project.

[4]More information can be found on Blender documentation webpage: http://www.blender.org/documentation/blender_python_api_2_63_14/info_gotcha.html

An example of an engine that provides intuitive class library and proper documentation is OGRE (**O**bject-Oriented **G**raphics **R**endering **E**ngine) - a 3D engine written in C++. OGRE is not a game engine, but it can be combined with other libraries to achieve desired functionality. OGRE has an active community and is well-known for good object-oriented design and flexible class hierarchy. Though OGRE itself does not provide any libraries for creating and animating human characters, character models can be imported from other sources. Blender 3D models can be exported to OGRE binary format - .mesh.

3.2.2 Exporting from Blender to OGRE

The main difference between data representation in Blender and OGRE is coordinate systems mismatch, as shown on Figure 3.3.

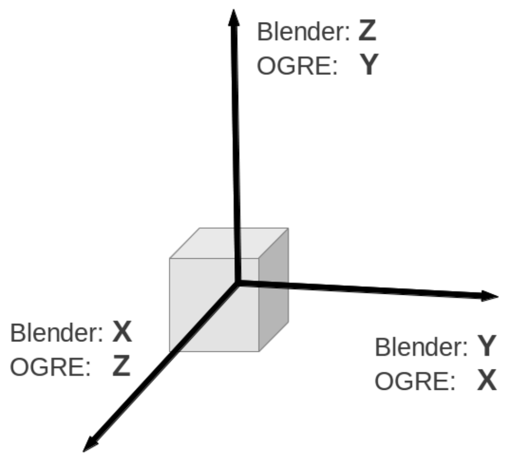


Figure 3.3: Difference between Blender and OGRE coordinate systems

Exporting from Blender format (.blend) to OGRE mesh format (.mesh) can be done using a Blender add-on called *blender2ogre*. The add-on supports conversion of both static and animated mesh objects. Exporting animated mesh objects requires some modifications to be done to them in Blender:

- An object with an armature is required to have zero location/rotation/scale transformation. Any transformations on the mesh or the skeleton need to be applied before adding an armature modifier.
- The maximum supported amount of bones in a skeleton is 256.
- The root bone(s) need(s) to have zero transformation on location and rotation (can be checked in edit mode in Blender).

- Each vertex can have blend weights for a minimum of 1 and a maximum of 4 bones.

- Animations need to be converted to non-linear animation (NLA) strips[5].

Both original and unofficial versions of *Mancandy* have more than 256 bones, including several root bones. To meet the export requirements, the amount of bones has to be reduced, and all the roots bones need to be adjusted respectively.

We have considered different solutions to meet the necessary requirements. Firstly, we tried reducing the total number of bones to exactly 256, which made it possible to export the armature with minimal modifications. Still, the requirement for bone weights was not fulfilled. *ManCandy* has vertices weighed to more than four bones. Increasing the value of Trim-Weights threshold option in *blender2ogre* reduced the number of bones per vertex, but it resulted in some vertices left without any bone assignments and caused wrong deformation of the mesh in animations. The problem occurred due to Blender supporting larger amount of bone weights assigned per vertex than OGRE.

Another solution we considered was creating a new rig with a simpler structure, which allowed assigning each vertex to not more than 4 bones in Blender. The mesh was acquired from the original *Mancandy* rig by exporting it to a common 3D mesh format (we used .obj format) and then, importing it into an empty Blender project. Since weights assignment is usually done automatically by the armature modifier in Blender, we concluded that is was better to use a basic armature with a minimal number of bones. Generally, an armature can be obtained from some source or created from scratch.

We have observed that it was very important to make all the necessary adjustments to the created rig before exporting it. Failure to satisfy any of the exporter's requirements can lead to unexpected and undesirable results. Figure 3.4 shows an example of mesh taken from *Mancandy*, and armature, adapted from a 3D model generated with Makehuman[6], an open source tool for making 3D characters. Armature was scaled to match the size of the mesh; the size and position of some bones were adjusted, and the values of the root bone were reset to meet the requirements. However, the scale and location transformations of the mesh and the armature were not applied, which failed to meet one of the exporter's requirements. Figure 3.4(a) shows that the model is represented correctly in Blender, whereas on Figure 3.4(b) we can see how the model looks in OGRE after being exported. Each set of three arrows on the figure represents a bone. Imagining a line that goes through the origin of each arrow set gives us a visual representation of the skeleton and makes it obvious that the skeleton is positioned incorrectly in relation to the mesh.

Creating a simple armature from scratch and parenting it to the mesh imported from

[5]Blender user manual explains usage of Non-Linear Animation Editor in more detail: http://wiki.blender.org/index.php/Doc:2.6/Manual/Animation/Editors/NLA

[6]http://www.makehuman.org/

ManCandy was our final solution. One should remember that the root bone of the armature should have zero transformation on location and rotation, while the position of the mesh needs to be adjusted accordingly. The total amount of bones cannot exceed 256. Finally, bone sizes have to be adjusted to distribute the weights properly, if the automatic weights assignment is used. In practice, bigger bones are needed for larger mesh parts. Parts of the mesh that are supposed to be more flexible require a greater number of bones. Areas where the mesh has a complex structure (larger amount of polygons) might need several bones, bigger bones or application of other methods to achieve proper weights distribution.

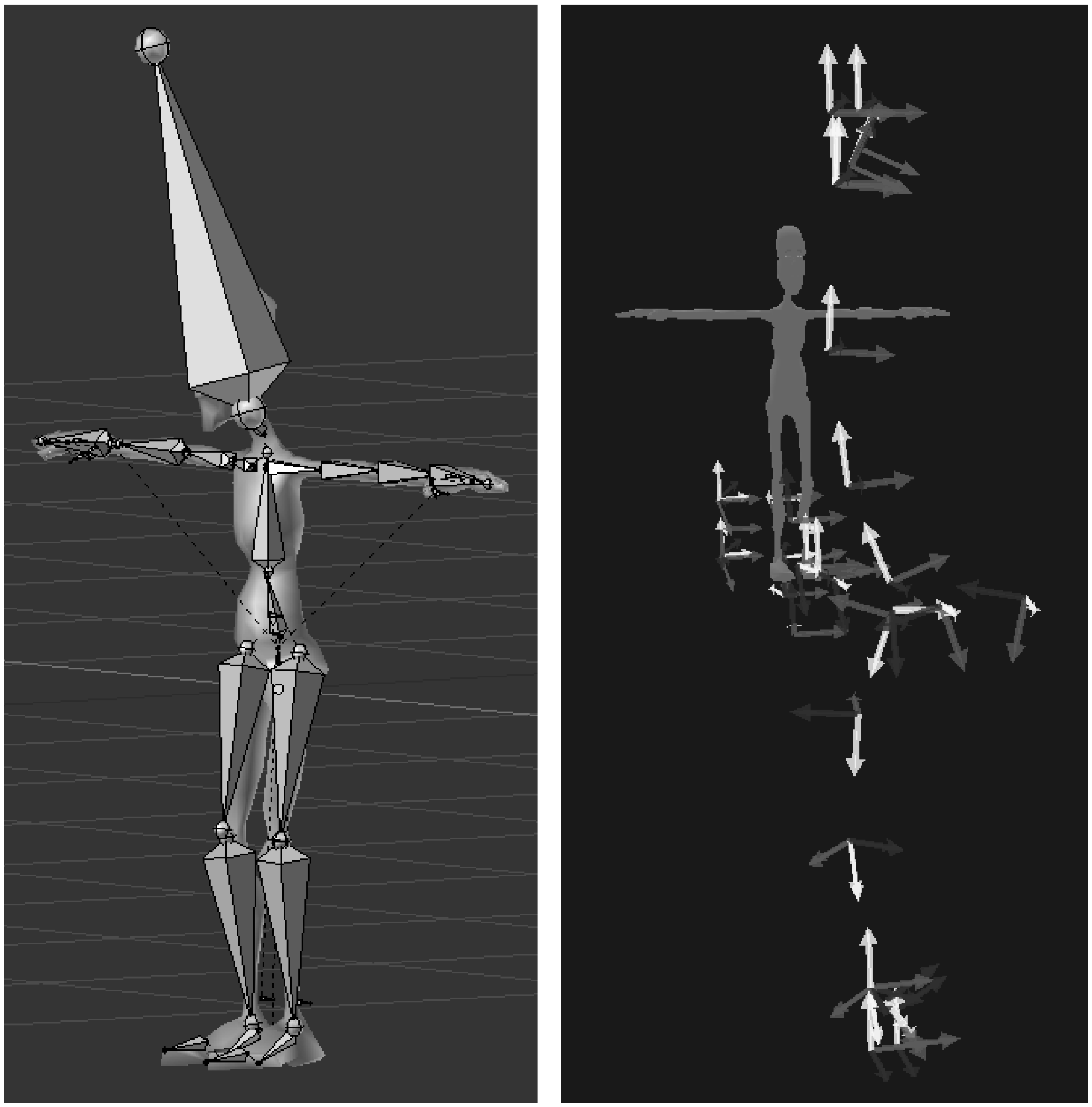

(a) Blender view before export (correct) (b) OGRE view after export (wrong)

Figure 3.4: Example of Blender to OGRE export failure

Figure 3.5 shows a basic armature consisting of 50 bones that we created in Blender, with the mesh borrowed from *ManCandy*. To obtain a more detailed view, the bones are shown by using different bone representations - octahedral (Figure 3.5(a)) and stick (Figure 3.5(b)). One can start wondering why the head bone is that large. The reason is that the head is represented as a complex mesh structure, and a larger bone is a simple solution used to achieve correct weights distribution. The size of this bone could have been smaller, but then, the weights would have to be adjusted in a different way. Since bones are not visible during run-time, using a large bone is not a problem, because it will not affect the visual representation. Another possible solution here would be using more than one bone for the head, but it could add unnecessary complications and would be beneficial only if facial animation was desired. Figure 3.5(c) shows that exporting the rig to OGRE was successful.

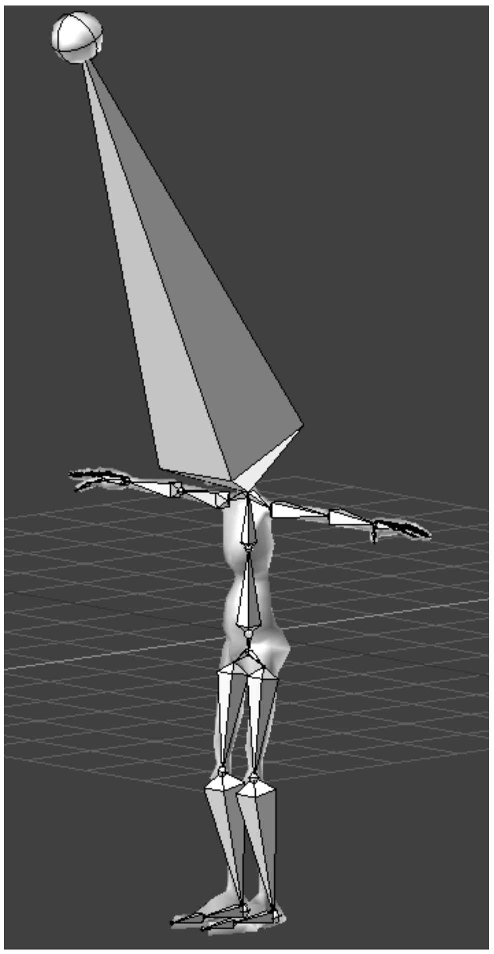
3.2.3 Methods of Creating Animations

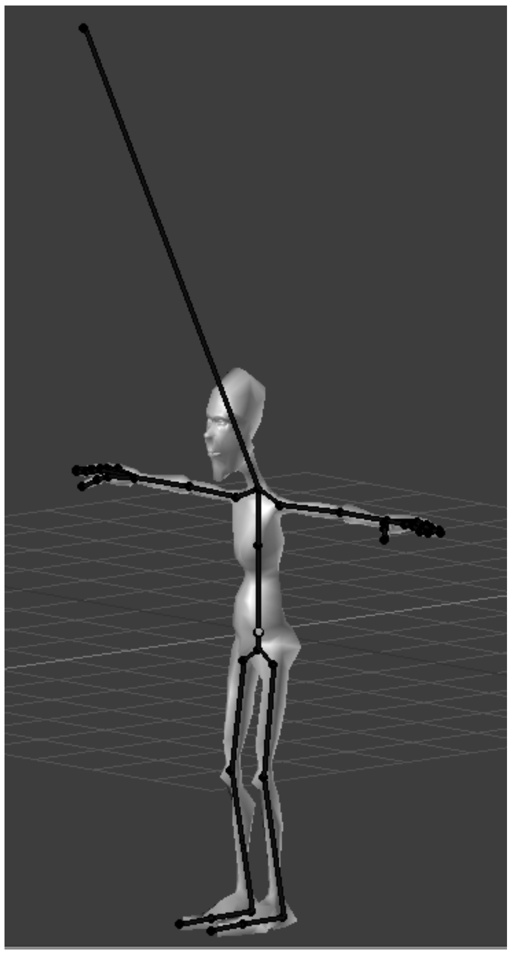
In a game model, where human characters are used as avatars, animations usually simulate human interactions. Typically, such interactions involve human articulations, gestures, and actions. Simulating realistic human motions has been a popular research topic during the past years, since human figures are widely used not only in computer games, but also, in many other applications. The techniques used for simulating human motion can be differentiated as automated methods, involving motion capturing, and manual animation methods.

Motion capturing is generally based on obtaining data from sensors attached to certain points on a human body. The locations of sensor placements and the type of information collected from the devices varies, depending on the technique applied. A method that is quite often used for determining sensor placement points is dividing human body into segments, such as head, torso, upper arm, elbow, etc. To reduce error rate and possible overhead in calculations, the amount of segments is usually minimal. The points that join these segments are defined as key points and are used to attach sensors to. Data collected from a sensor usually contains a vector describing sensor's position and a quaternion describing its rotation.(Li et al., 2009). This information can be then used to change the position and rotation of the corresponding bone on the virtual human model. Animations generated in such way can effectively simulate human articulations or actions, but the method itself requires a large amount of resources, including motion sensors and a system of data processing.

Manual creation of animations is usually performed by using 3D modeling software. The most common method is translating and rotating skeleton bones of the animated character and storing changed data in key frames. Interpolating between the key frames generates animations. The method does not require usage of any specialized equipment, such as motions sensors, which is certainly an advantage. Nevertheless, creating realistic animations manually is rather time-consuming.

Animating a human figure with a simple skeleton can be done in both described ways. However, the first method might be expensive due to required equipment and data


(a) Skeleton shown as octahedral


(b) Skeleton shown as stick

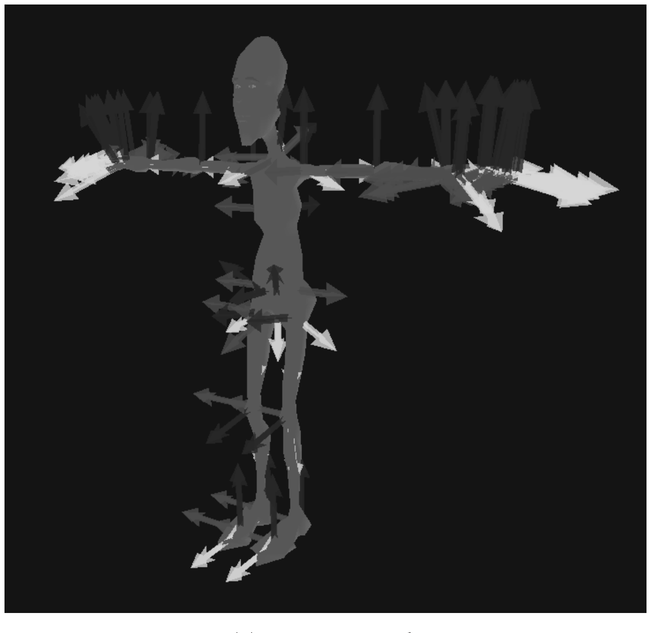

(c) Export result

Figure 3.5: Basic armature constructed from scratch in Blender

processing. Besides, our game model does not allow capturing player movements using sensors. It could be helpful to apply motion capturing for obtaining a motion database and use it for generating animations, but it would not influence the research outcome significantly. Therefore, we consider creating animations manually to be an optimal solution.

Since *ManCandy* was created in Blender, it is adequate to use Blender for making animations. Animating a character in Blender is done in Pose Mode, using Action Editor. A separate action needs to be created for each animation. Character's skeleton is manipulated with FK or IK, and data about each bone manipulation is stored in a key frame. A detailed overview of the animation process is covered in Blender user manual. [7] For exporting an animation from Blender to OGRE, it is necessary to convert the action to an NLA strip, as already mentioned in Section 3.2.2.

3.2.4 Categories of Player Interactions

Quality of a multiplayer game significantly depends on the success of player interactions. What we refer to as 'success' means meeting players' expectations and providing a smooth and enjoyable gaming experience in general. In any case, most players would expect an almost immediate response when they trigger an action by using input devices, such as keyboard or joystick. In other words, when a player presses a button, they expect something to happen. For example, any typical player action, such as walking, running, jumping or shooting is expected to start right after a player triggers it. However, since most online multiplayer games implement a client-server model, the server typically plays an authoritative role in simulating the game. That is, all the actions performed by a client need to be acknowledged by the server.

If clients have to send requests to the server and wait for acknowledgements before processing events, a noticeable delay is impossible to avoid, unless the game is played on LAN. A large delay between triggering an event and its actual execution often leads to player dissatisfaction and frustration. It is one of the main reasons why multiplayer games generally implement a strategy, where game events are executed immediately after they are triggered by the user, assuming temporarily that the server will acknowledge them. In case if the simulation state of the client becomes different from the one simulated by the server, the client's state is corrected upon receiving a server response. Performing such corrections efficiently becomes a harder task as the latency increases. Firstly, the correction can only start taking place in at least a full round trip time between the client and server. Secondly, correcting critical actions is not always possible to do in an unnoticeable to the user way.

As discussed in Chapter 2, different types of game events are not equally sensitive to increased latency. Performing simple actions that do not affect the states of other players is usually the easiest task. For this reason, it was observed that most RTS and RPG

[7]http://wiki.blender.org/index.php/Doc:2.6/Manual/Animation

games, based mainly on strategic thinking and skills development, but not on fast-pace interactions between the players, could generally tolerate rather high latencies. In contrast, studies based on FPS games and racing simulators concluded that an increase in latency caused degradation of user performance. As already mentioned, acknowledging user actions immediately is obviously impossible due to the network delay. However, the time is takes to process an acknowledgement really matters. Moreover, we believe that the tolerable latency threshold is not the same for different types of player interactions.

Consider an example of a game, where avatars are represented by human characters and can perform such actions as shaking hands, punching each other, playing a game called "Rock-paper-scissors", and waving at each other. Each action is interactive, meaning that two players are involved. Ideally, performing such interactions would require precise information about players' kinematic states, but that is impossible due to the network delay. If we execute an action locally on the client, the client application would use the latest known data about players' states to decide whether execution of the action is possible. That is, whether the players are close enough to each other to do a punch, a handshake, to play "Rock paper scissors", or whether they can see each other, which is necessary to perform a wave. In case the client decides that execution of an action is possible, its simulation starts immediately, while a verification request is sent to the server at the same time. The server checks whether the positions of both players indeed satisfied the action's requirements at the time when it was initiated, and decides whether the action is acknowledged or not. If the server acknowledges the action, the action request is also forwarded to the other players. If not, then the client that committed it corrects the score accordingly, after receiving the server's message.

Delayed delivery of updates can prevent the clients from continuously having recent kinematic state information about remote players, which is needed to decide whether a certain player action is legal or not. Due to the presence of latency, clients can sometimes make wrong simulations and eventually perform necessary corrections, affecting players' score results negatively. We suppose that at high latency, scoring an action that requires players to be located within short distance range can become more difficult; nevertheless, performing actions that do not have strict requirements to players' locations might still be feasible. To verify our hypothesis, we implement two categories of players interactions - **close-range interactions** and **long-range interactions**. Inspired by findings discussed in Chapter 2, we use a third person avatar perspective, since we believe it is most suitable for representation of human interactions, allowing the users to have a complete view of avatars' actions.

The short-range interactions (Figure 3.6) include:

Punch: one player punching another player. Players need to be close to each other for the action to take place.

Handshake: two players shaking each others hands. Handshake requires that the distance between the players does not exceed a certain value, though the distance limit is slightly higher than in case of punch.

The long-range interactions (Figure 3.7) include:

Rock-paper-scissors: two players playing a hand game. The game is also referred to as **roshambo** and enables each player to choose one of the three items - rock, paper or scissors. Clenched fist represents rock, two extended and separated fingers represent scissors, and an open hand represents paper. (Fisher, 2008) In our case, the choice is not made by the player, but determined randomly instead, when the action is initiated. Rock-paper-scissors requires that the players are able to see each other well enough to be able to recognize the item choice, but the distance limit is relatively large.

Waving: a player waving hand to another player. The only requirement for this action is the ability of both players to see each other.

In addition to the actions listed above, we differentiate between two possible player states, taking place when no score-earning action is performed:

Walking: gives a player the possibility to move forward or backward or/and turn left or right.

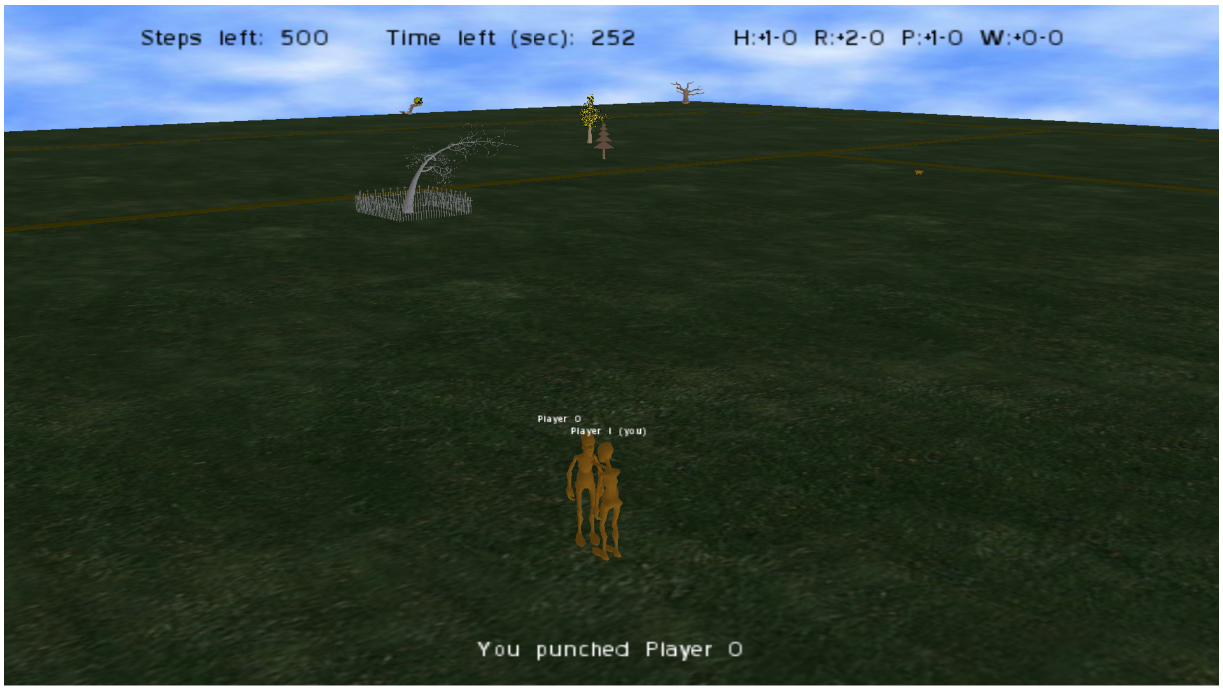
Idle: takes place automatically, when the latest triggered action was fulfilled, no user input was done, and no response to other player's actions is needed.

To express each action's distance constraints and walking speed numerically, we use OGRE's coordinate system units which we refer to as "steps". Considering that each point in the coordinate system is represented by a vector, the distance between two vectors can be expressed as a number. For example, the distance between two vectors (0,0,0) and (0,0,1) is equal to 1 (we refer to it as 1 step). The Table 3.1 lists the characteristics of for each action.

Table 3.1: Actions summary

Action	**Earns score**	**Range**	**Distance/Speed**	**Other Constraints**
Punch	yes	short	150 steps	500 steps away from last score-earning action's location
Handshake			250 steps	
Rock-paper-scissors		long	1000 steps	
Wave			-	
Walk	no	other	450 steps/sec, 35 degrees/sec	absence of collision
Idle			-	-

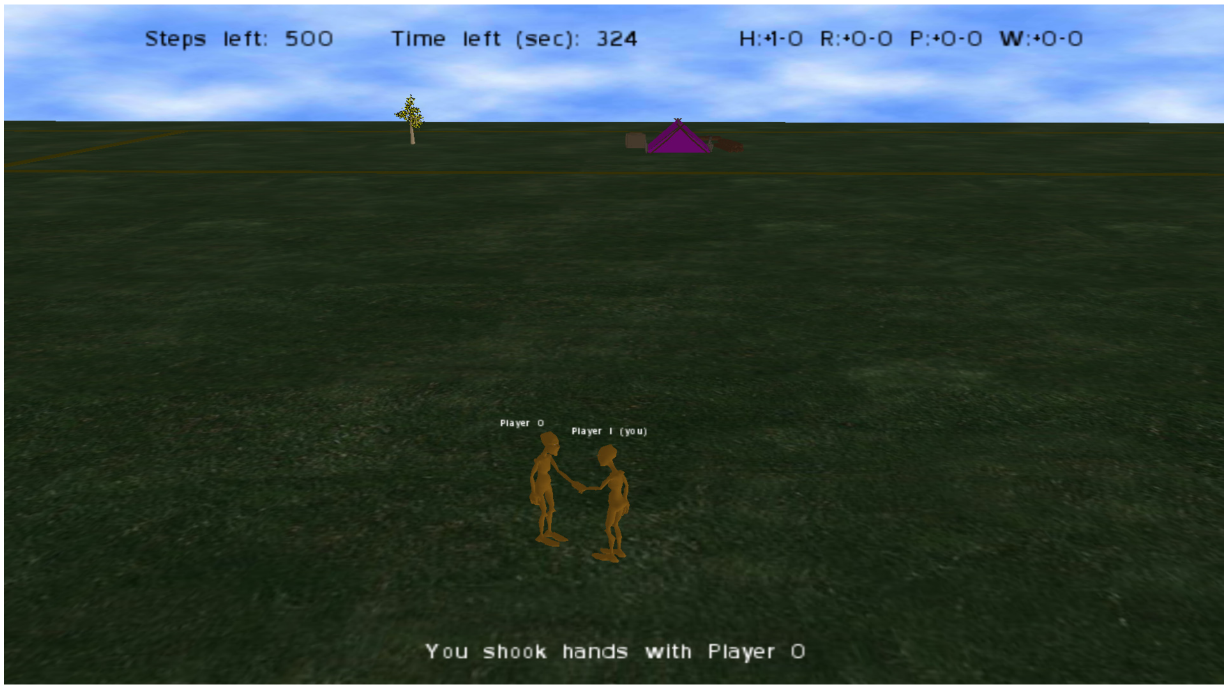
Players can walk with the speed of 450 steps per second and turn as fast as 35 degrees per second. The walking speed is constant as we do not use any acceleration. Each score-earning action has an additional constraint - a player needs to walk 500 steps



Steps left: 500 Time left (sec): 252 H:+1-0 R:+2-0 P:+1-0 W:+0-0

You punched Player 0

(a) Punch

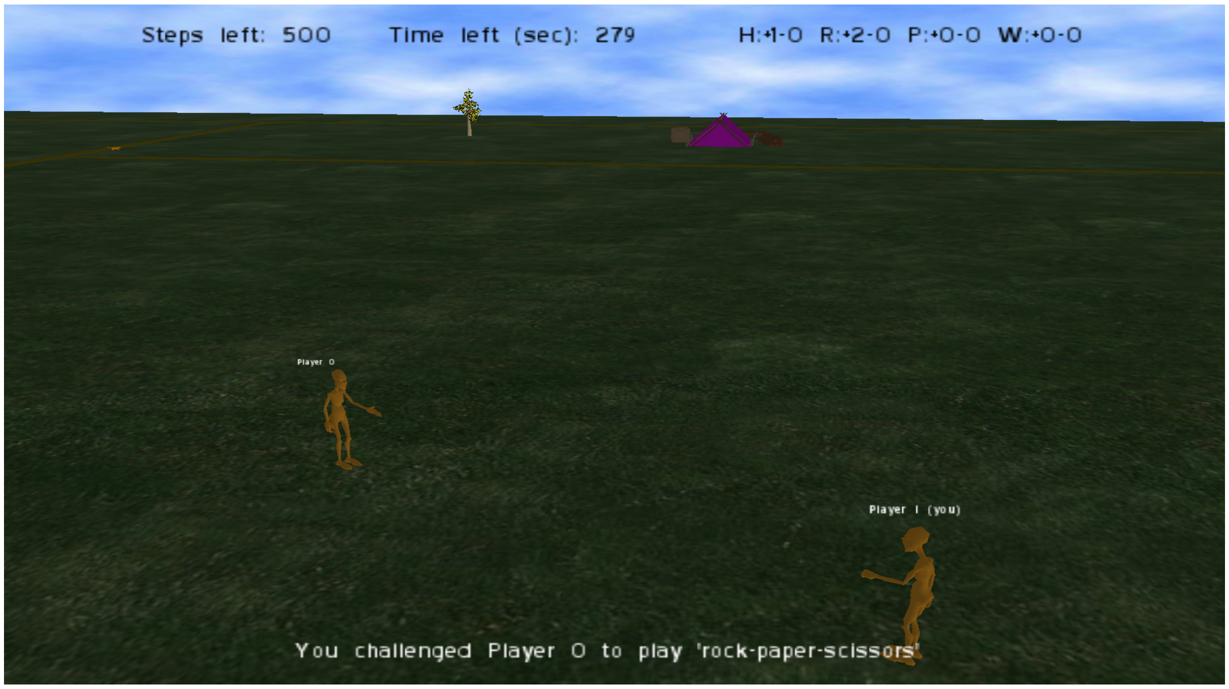


Steps left: 500 Time left (sec): 324 H:+1-0 R:+0-0 P:+0-0 W:+0-0

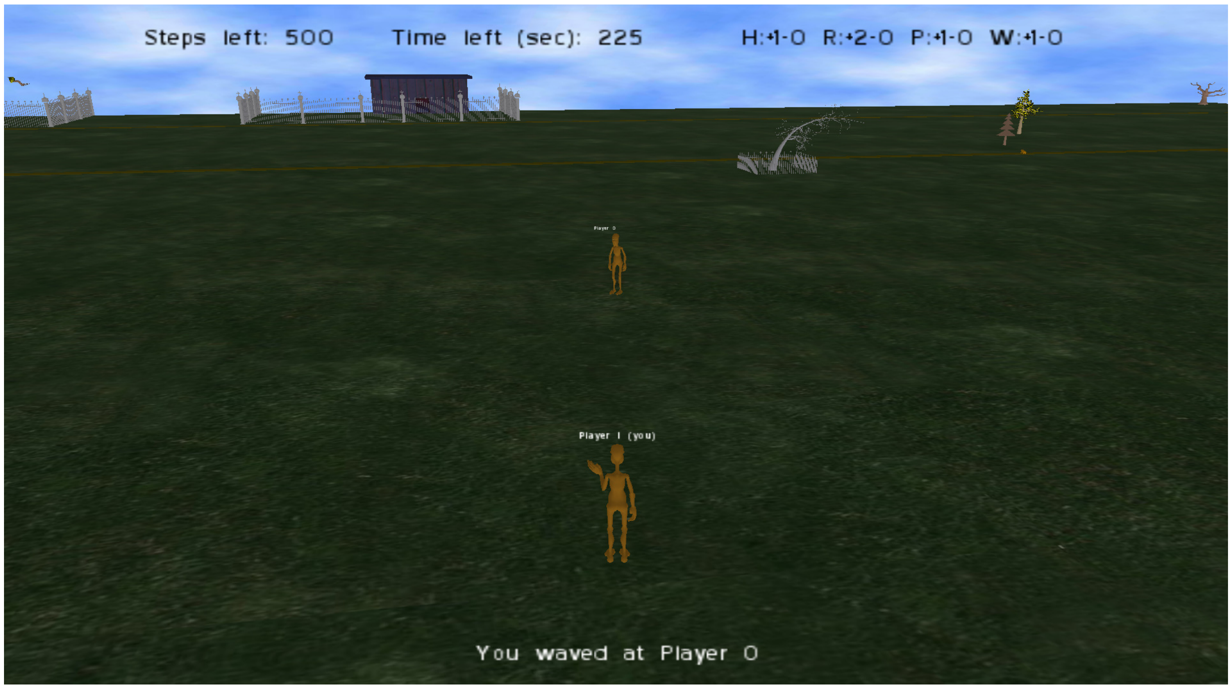
You shook hands with Player 0

(b) Handshake

Figure 3.6: Short-range player interactions


Steps left: 500
Time left (sec): 279
H:+1-0 R:+2-0 P:+0-0 W:+0-0
Player 0
Player 1 (you)
You challenged Player 0 to play 'rock-paper-scissors'

(a) Rock-paper-scissors



(b) Wave

Figure 3.7: Long-range player interactions

away from the point where the last score-earning action was performed. This needs to happen at least once after an action was scored, and after that, the player can perform another action at any desired location. This requirement is introduced to motivate the players to change their positions each time a new critical action is executed. If players' locations do not change, scoring an action becomes simpler, since old positions remain valid for longer, which in turn reduces the likeliness of inconsistency. Passive behavior is usually not expected from players in interactive multiplayer games, and we would like to eliminate such cases from our analysis. In other words, we are interested in considering the situations where inconsistencies in clients' simulations of the game occurred, but not the cases where such inconsistencies were very likely, but did not take place due to players' inactivity.

For obtaining a statistically useful score outcome, we define the goal of our test game: to win, a player has to perform each score-earning action at least 10 times. The game ends when at least one player has achieved the goal. Also, we introduce a time limit of 350 seconds for each game round, to prevent users from playing endlessly.

3.2.5 Physics Simulation and Collision Detection

Many modern games use physics laws in simulating virtual world. The purpose of that is making the game world look more real to the users. Physics simulation is usually performed only as an approximation to the real world physics, and discrete values are used in computations. Such simulation is typically implemented in **physics engines** - software, providing the simulation of different physical systems, such as **rigid body dynamics**, which defines the movement of object under application of external forces, **soft body dynamics**, focusing on soft and deformable objects' motion, and **fluid dynamics**, generating realistic animations of liquids, smoke, and relative substances. Physics engines are generally categorized as **real-time** and **high-precision** engines. High precision engines aim at simulating physics rather precisely and therefore, require a lot of computational power. Such engines are typically used in scientific simulations and computer-animated movies. Video games and other interactive applications use real-time engines, which are based on simplified calculations and somewhat decreased accuracy, which allows to speed up the simulation and maintain an appropriate gameplay rate. (Eberly, 2003)

Collision detection is rather useful part of physics processing in real-time engines, as it often defines how game objects interact with each other. Most 3D objects in a virtual environment that includes physics simulation are represented by two separate meshes. The main mesh is rather complex and detailed. Since it is used for rendering, it represents the visible part of the object. For physical computations, it is typical to use a secondary, simplified mesh, which only defines objects' shape and properties and is usually referred to as **collision geometry**. Examples of collision geometry are **bounding box** - the smallest box that encloses the object, **sphere** or **convex hull** - the smallest set of points in the Euclidean plane or space that contains the object. A simple way to detect a collision of two separate objects is to check whether their collision geometry meshes intersect. Generally, collision detection is performed as a process involving two phases: **broad phase**, where the main task is to simplify computations for objects that are far away from each other, and **narrow phase**, where each individual pair of objects is carefully checked for collision. (Ericson, 2005)

A good example of a popular real-time open-source physics engine is Bullet[8]. Its applications include physics simulation in games and producing visual effects for movies. It is integrated in BGE and can also be combined with OGRE. For the test game model that we implement, using Bullet physics for collision detection is a possible option. For this reason, our first solution idea for implementing physics simulation and collision detection in our test model was based on using Bullet. To combine Bullet with OGRE, we would need to convert some data (for example, meshes) from OGRE format to the format used by Bullet and back. There are wrappers that have that functionality, and one of the most popular ones is OgreBullet[9]. For the sake of simplicity, we decided to use OgreBullet to integrate Bullet engine into our project.

It is necessary to mention that Bullet provides quite extensive functionality, including collision detection, simulating motion of rigid bodies, and dynamics of soft bodies. The only feature that would be useful for us is collision detection, since simulating realistic physical world is not that important in our work. To make use of Bullet's collision detection library, we need to simulate a collision world, which can be done in the following way:

```
// gravity vector for Bullet
Vector3 gravityVector = Vector3(0,-500,0);
// Bullet dynamics world wrapped by OgreBullet
OgreBulletDynamics::DynamicsWorld *mWorld;
mWorld = new OgreBulletDynamics::DynamicsWorld(mSceneMgr,
   AxisAlignedBox
      (Vector3 (-10000, -10000, -10000),Vector3 (10000, 10000,
         10000)), gravityVector);
```

The physics animation needs to be updated every time a new frame is rendered:

```
mWorld->stepSimulation(evt.timeSinceLastFrame);
```

Each time we create an object with physical properties, we need to add a rigid body object and assign some collision shape to it. That is useful for non-animated objects, that do not change their shape over time. Consider a very simple example: we would like our avatar to be able to throw balls and implement it as follows:

[8]More information about Bullet can be found on the official website: http://bulletphysics.org/wordpress/

[9]http://www.ogre3d.org/tikiwiki/tiki-index.php?page=OgreBullet

```
// starting position of the ball - avatar's position + 10
Vector3 position = localPlayer->playerNode->getPosition() + 10;
// starting orientation of the box
Quaternion orientation = localPlayer->playerNode->getOrientation();

// create OGRE mesh for the ball
Entity *entity = mSceneMgr->createEntity(
   "Ball" + StringConverter::toString(mNumEntitiesInstanced),
   "Sphere.mesh");
entity->setCastShadows(true);
entity->setMaterialName("Examples/BumpyMetal");

// we need the bounding box of the ball to set the size of the
   Bullet container
AxisAlignedBox boundingB = entity->getBoundingBox();
SceneNode *node =
   mSceneMgr->getRootSceneNode()->createChildSceneNode();
node->attachObject(entity);
// the ball is too small for us, make it bigger
node->scale(10, 10, 10);

// approximate radius of the ball, adjusted manually
Real radius =
      (boundingB.getCenter().distance(boundingB.getMaximum()))*6;

// create the Bullet sphere shape with the calculated radius
OgreBulletCollisions::SphereCollisionShape *sceneShape =
      new OgreBulletCollisions::SphereCollisionShape(radius);

// and the Bullet rigid body
OgreBulletDynamics::RigidBody *defaultBody =
      new OgreBulletDynamics::RigidBody("defaultRigid" +
      StringConverter::toString(mNumEntitiesInstanced),
      mWorld);

// set physical properties of the ball
defaultBody->setShape(node,
               sceneShape,
               0.0f,       // dynamic body restitution
               100.0f,     // dynamic body friction
               25.5f,      // dynamic bodymass
               position,
               orientation);
mNumEntitiesInstanced++;

// speed of the ball's motion when it is thrown
defaultBody->setLinearVelocity(position.normalisedCopy() * 7.0f);
```

```
// push the created objects to the deque, so that we can delete them
   at exit
mShapes.push_back(sceneShape);
mBodies.push_back(defaultBody);
```

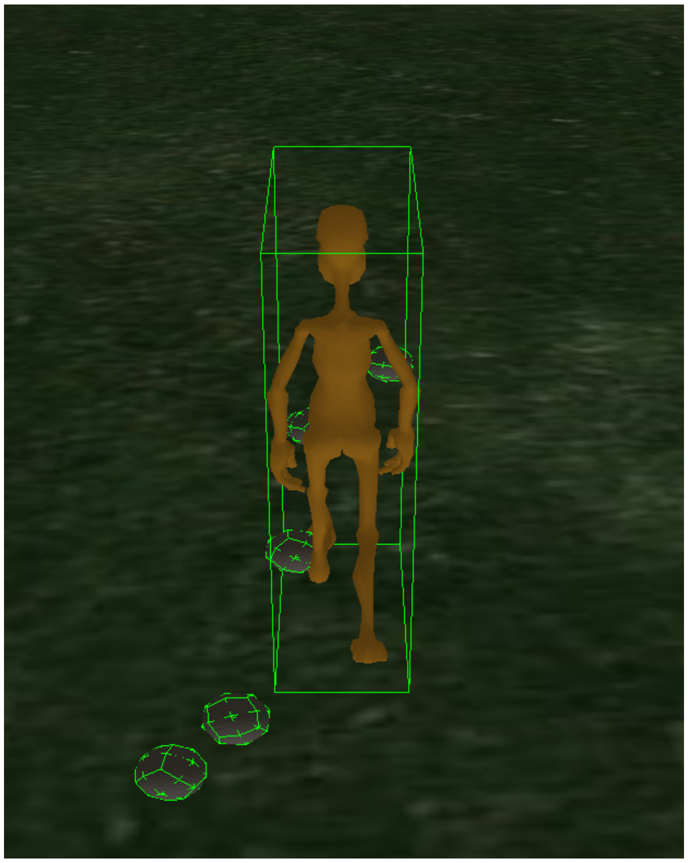


Figure 3.8: Using Bullet with OgreBullet for collision detection: collision shapes

As a result, created balls behave correctly in the physical world (Figure 3.8). We set a Bullet container around the avatar, in a similar way we did with the ball objects, which makes it possible for the avatar to interact with the physical objects (in this example - balls). If the player collides with a ball, the ball gets a physical impulse and moves in the corresponding direction. If a ball falls on the avatar, it bounces off. However, the avatar itself is not be affected by balls' movements. We do not want player object's motion to be defined by other physical objects, since it is controlled by user input. Therefore, we set the avatar's rigid body to be kinematic, which means it can affect other physical bodies, but not vice versa:

```
playerRididBody->setKinematicObject(true);
```

While simulating physics, as in the example discussed, creates nice effects in a game world in general, we do not have a need for such functionality in our test game model. What we do need is detecting collision of an avatar with another avatar, since players should not be able to walk through each other, and also, with some static world objects.

Originally, we intended to use Bullet for narrow-phase collision, creating shape containers around avatar's bones. In that way, the collision containers would follow the bones during animations, which could be a rather useful feature when dealing with human interactions. OgreBullet has a class that performs conversion of animated mesh to Bullet format and can be used to make containers for bones of an animated entity, in our case - avatar. That can be done in the following way:

```
//create a converter to make containers for animated mesh
OgreBulletCollisions::AnimatedMeshToShapeConverter *cont =
      new OgreBulletCollisions::AnimatedMeshToShapeConverter(
      localPlayer->playerEnt,
      *(localPlayer->playerEnt->_getBoneMatrices()));

// get pointer to the avatar's skeleton instance
SkeletonInstance *skel = localPlayer->playerEnt->getSkeleton();

int numBones = skel->getNumBones();
int i;

// skip the root bone which has handle 0
for (i = 1; i < numBones; i++)
{
   Bone* bone = skel->getBone(i);

   //create a SceneNode for the collision shape as a child of
      avatar's SceneNode
   SceneNode *node =
      mSceneMgr->getRootSceneNode()->createChildSceneNode(
      StringConverter::toString(mNumEntitiesInstanced),
      bone->_getDerivedPosition());

   // create capsule collision shape for each bone
   OgreBulletCollisions::CapsuleCollisionShape *boneShape =
      cont->createOrientedCapsuleCollisionShape(
      bone->getHandle(),
      bone->_getDerivedPosition(),
      bone->_getDerivedOrientation());

   // add Bullet rigid body
   OgreBulletDynamics::RigidBody *boneRigidBody =
      new OgreBulletDynamics::RigidBody(
      "defaultBoxRigid" +
         StringConverter::toString(mNumEntitiesInstanced),
      mWorld);
   // no physical properties are needed, set only position and
      orientation
   boneRigidBody->setShape(node,
               boneShape,
               0.0f,
               0.0f,
               0.0f,
               bone->_getDerivedPosition(),
               bone->_getDerivedOrientation());
   mNumEntitiesInstanced++;

   // we do not want the body's motion to be affected by physics
   boneRigidBody->setKinematicObject(true);

   // push the created objects to the deque for deleting at exit
   mShapes.push_back(static_cast
      <OgreBulletCollisions::CollisionShape *> (boneShape));
   mBodies.push_back(boneRigidBody);
}
```

As we can see on Figure 3.9, we created a container for each bone, but the result was not satisfactory. First of all, the containers did not follow the bones during animations, as we expected. Instead, they seemed to stay on bones' original positions (before any animation was applied). In addition to that, not all the collision shapes had correct sizes and rotations. The container for the right hip bone was placed and rotated correctly, while the container for the left hip bone had wrong rotation. One of the containers was shifted to the side, while all the others were too small (visible as small green dots in the background). We would not need a collision geometry for every single bone, but in any case, this solution did not give us the expected result.

Unfortunately, there is no official documentation for the OgreBullet project. Therefore, it is often unclear which objects need to be passed to its classes. For example, to create a converter for animated mesh, we need an entity and a transformation matrix:

```
AnimatedMeshToShapeConverter(Ogre::Entity *entity, const
   Ogre::Matrix4 &transform = Ogre::Matrix4::IDENTITY);
```

In the example dicussed above, we used bone matrix information as the transformation matrix. We also tried to use the full transformation of the entity's parent node instead:

```
OgreBulletCollisions::AnimatedMeshToShapeConverter *cont =
      new OgreBulletCollisions::AnimatedMeshToShapeConverter(
      localPlayer->playerEnt,
      localPlayer->playerEnt->_getParentNodeFullTransform());
```

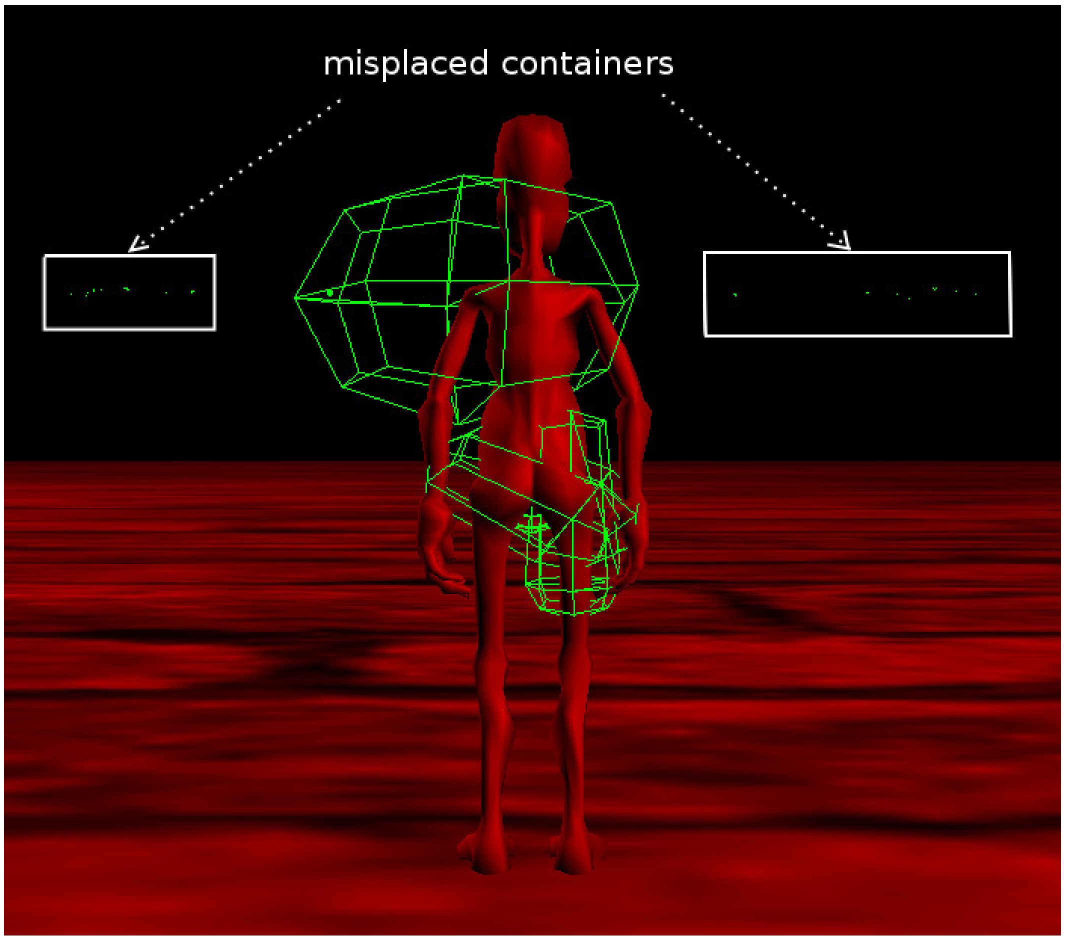


Figure 3.9: Misplaced collision shape containers (using bone matrix information for the animated mesh converter)

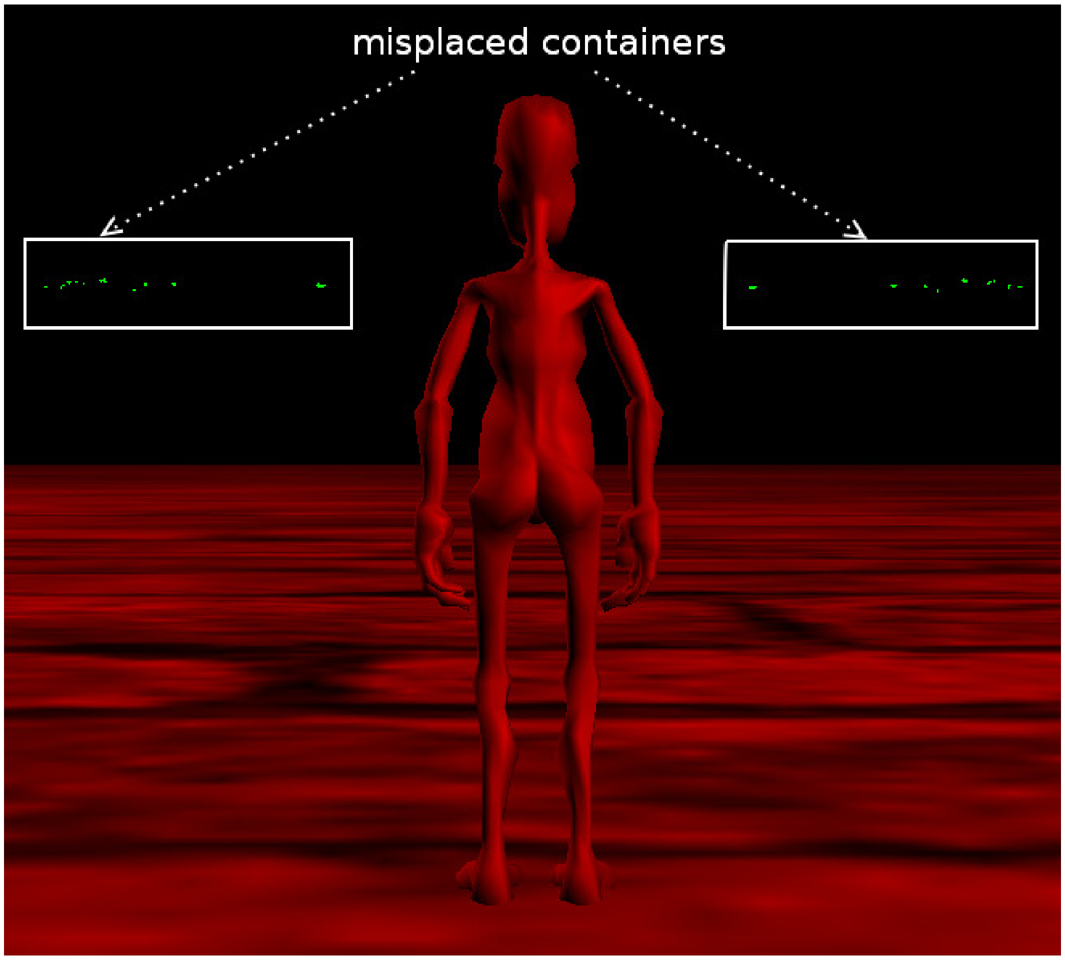


Figure 3.10: Misplaced collision shape containers (using full transformation matrix of the entity's parent node for the animated mesh converter)

The result was similar (Figure 3.10), except that all of the containers seemed to be too small.

To figure out why the solution did not produce expected results, we could approach the problem from different perspectives. One possibility would be to analyze the Ogre-Bullet's source code more thoroughly and spot possible reasons for the problem. As a different option, we could use another wrapper for Bullet, like BtOgre[10]. However, the problem could also originate from Bullet itself. When using some of the collision shapes, we got the following message at runtime:

> "Overflow in AABB, object removed from simulation. If you can reproduce this, please email bugs@continuousphysics.com
> Please include above information, your Platform, version of OS.
> Thanks."

In any case, since creating a simple collision geometry without taking animated mesh into account produces correct results, we consider that option to be more effective. An optimal way to implement collision detection for an avatar could be using Bullet's Kinematic Character Controller[11], since it provides functionality for controlling a game character and collision handling. However, there is no wrapper class for it in OgreBullet.

A simple way to perform broad phase collision with Bullet is calling Bullet's collision dispatcher directly:

```
// get the pointer to Bullet's dynamics world object
btDynamicsWorld *bulletWorld = mWorld->getBulletDynamicsWorld();
int numberOfManifolds = btWorld->getDispatcher()->getNumManifolds();
btSimpleBroadphase *simpleBroadphase = new btSimpleBroadphase();

for (int i = 0; i < numberOfManifolds; i++)
{
   btPersistentManifold* contactManifold =
      btWorld->getDispatcher()->getManifoldByIndexInternal(i);

   int numberOfContacts = contactManifold->getNumContacts();

   for (int j = 0; j<numberOfContacts; j++)
   {
      btManifoldPoint& contactPoint =
         contactManifold->getContactPoint(j);
      if (contactPoint.getDistance() < 0.f)
      {
         // collision: do something if necessary
      }
```

[10] https://github.com/nikki93/btogre
[11] http://www.continuousphysics.com/Bullet/BulletFull/classbtKinematicCharacterController.html

```
    }
}
```

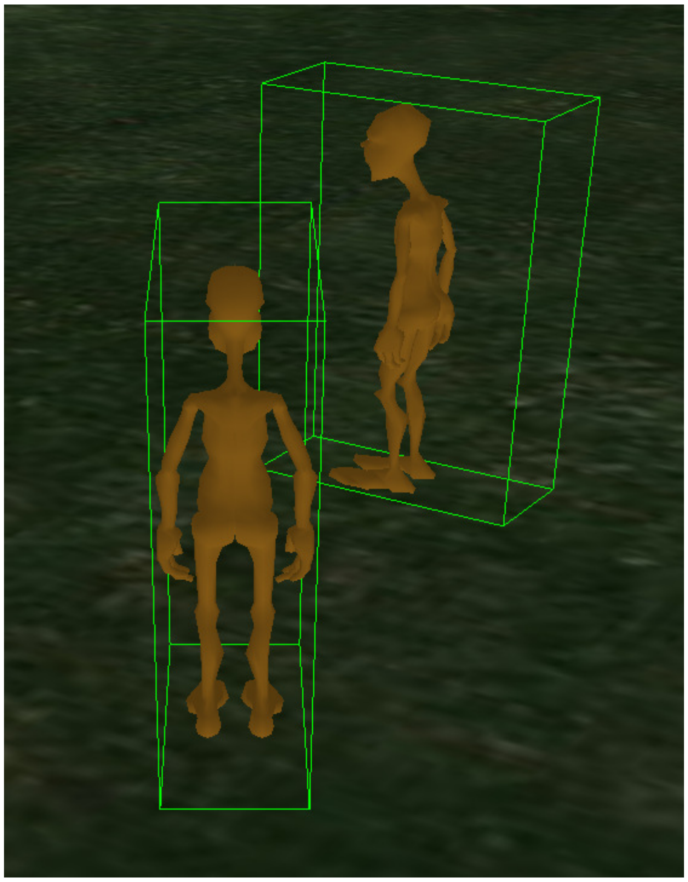


Figure 3.11: Example of simple collision geometry for avatars (using Bullet)

It is necessary to note that for determining exactly which objects collide, we need to keep track of all Bullet rigid bodies that are created by OgreBullet. Although, if we only want to detect collision occurence (for example, if we only add collision geometry to no other physical objects than avatars and want to know when they collide), then using the above algorithm is sufficient. An example of simple collision geometry that we considered to use for the avatar objects is shown on Figure 3.11. As we can see, the container around the avatar object is slightly bigger than the mesh, since we adjusted its size to the entity's possible animated states. For instance, if a smaller container size would have been used, then avatar's feet could move outside the box during walking. We considered this solution satisfactory, however, it required considerably more computational power and therefore, reduced application's responsiveness.

To avoid significantly decreasing the frame rendering rate by expensive collision detection computations, we decided to use essentially simplified collision detection method instead. Every time we change the position of a movable object (avatar) A in the virtual world, we calculate the distance between its new position and every other object it can collide with. Collision occurs if the distance between A and any other object exceeds

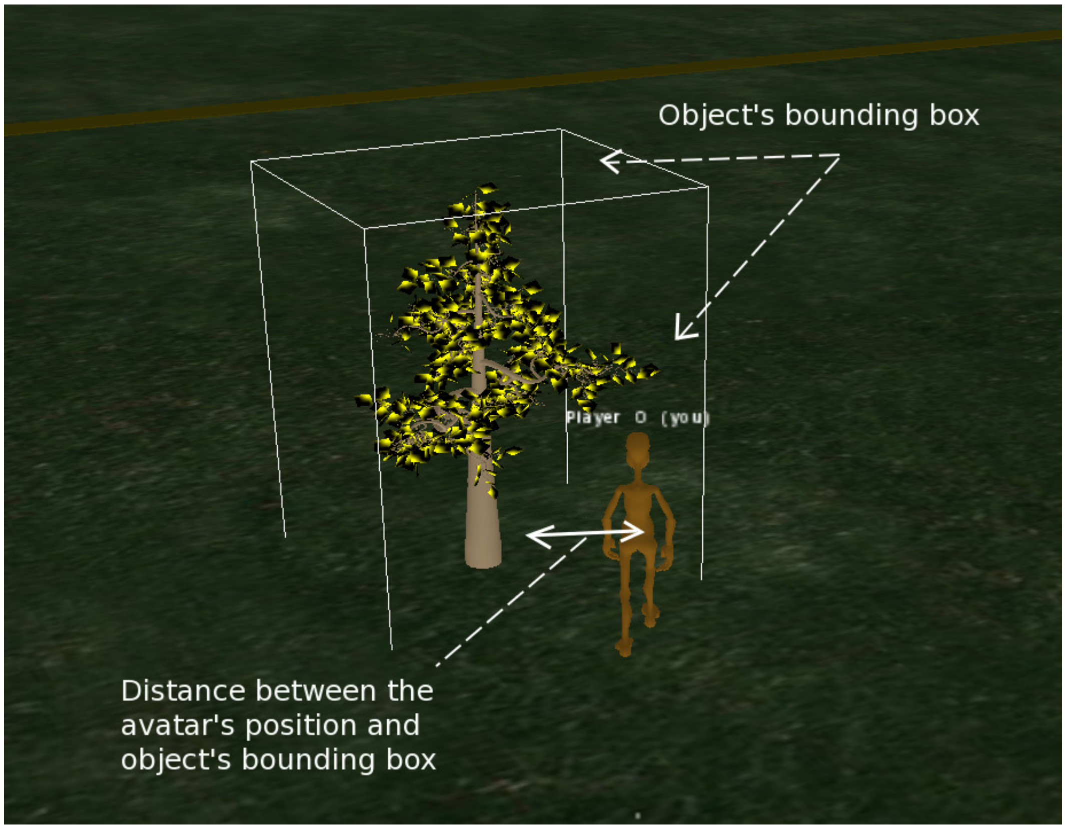

Object's bounding box
Player 0 (you)
Distance between the avatar's position and object's bounding box

Figure 3.12: Detection of avatar's collision with game world objects

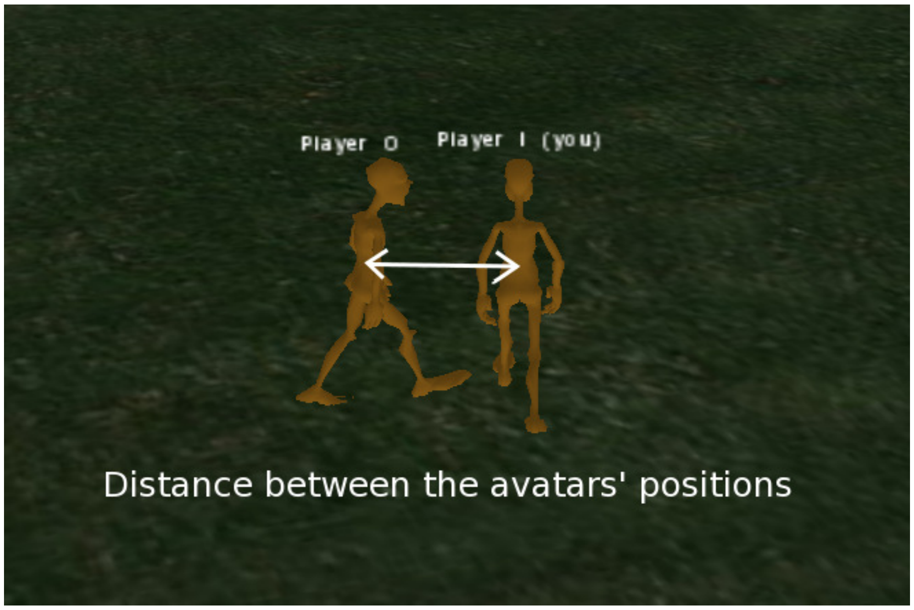


Figure 3.13: Detection of avatar's collision with another avatar

a certain limit. For detecting collision of A with any of the static game world objects, we calculate the distance between A's position and any point of the colliding world object's bounding box (Figure 3.12). To check whether avatars collide with each other, we only use their positions in calculations. We do not use avatar's bounding box geometry, since a bounding box typically encloses the object's mesh in its original state, before any animation is applied. Therefore, using the distance to avatar object's center is easier for adjusting calculations (Figure 3.13). If collision occurs, we move A back to its previous position.

Most physics engine also simulate gravity. In our game world, all dynamic objects can only move on a plane, along X and Z axes in OGRE's coordinate system, and never along the Y axis. Therefore, we do not need to simulate gravity and can only translate the objects to the required positions, according to user input or kinematic state updates.

3.3 Protocols

3.3.1 Client-Server Protocol

Most multiplayer networked games are based on a client-server model. Initially, game clients in early client-server game models served only for user input sampling, sending the commands to the server and processing the game simulation data received from the server. In that case, a client was basically responsible for sampling user input and rendering the objects, while the server was executing user commands and moving the objects in the game world. Such architecture is not suitable for modern video games played on the Internet, where users can experience some latency, since waiting for server acknowledgements is usually not acceptable. For this reason, most multiplayer games, particularly those based on Quake III and the Unreal Tournament engines, as well as their descendants, implement a modified client-server model. The implementation details of such architecture are usually adjusted, depending on the specific game model, but the basic principle is illustrated on Figure 3.14: each client process user input commands, executes them locally, assuming for a while that the server will acknowledge them. The server receives clients' messages, executes the commands, simulates the world accordingly, and sends the outcome to the clients. Each client corrects the simulated state of the game world, if necessary. (Bernier, 2001)

The client-server protocol that we implement in our test game model is based on a similar principle - there is an authoritative server and clients connected to it. The server is running several concurrent threads:

- `Main`: continuously accepts new connections;

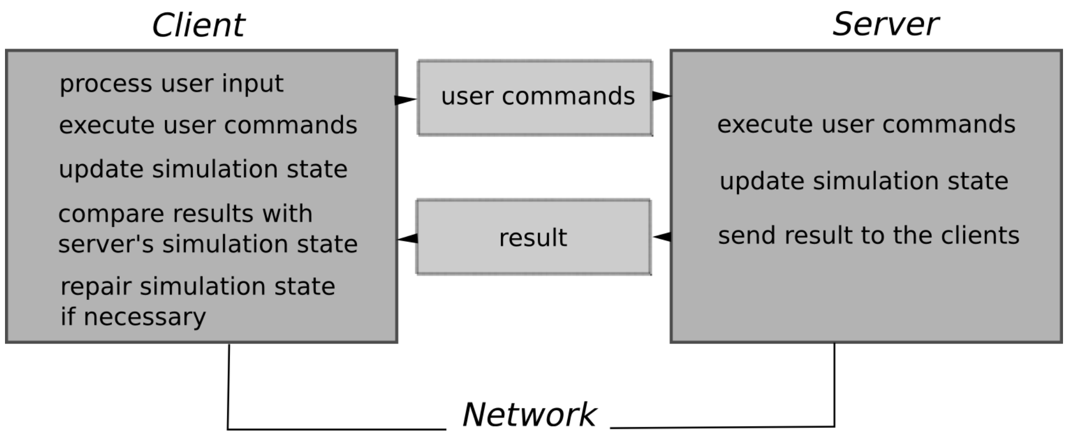


Figure 3.14: Basic client-server architecture

- `SocketReader` (separate thread for each client): receives messages from the client;

- `BufferHandler`: processes the messages received by clients, packages them into a single buffer, and sends an update to each client;

The client implementation is based on four concurrent threads:

- `Main`: manages the frame rendering loop (simulates and renders the game world), processes user input;

- `Sender`: sends messages to the server;

- `Receiver`: receives server updates;

- `Delay simulator`: simulates specified network latency by delayed processing of received messages;

The `Main` thread on the server accepts connections from clients and creates a separate TCP socket for every client that joins. We use TCP because we need reliable delivery of all messages. To start a game, a client establishes connection with the server. The server assigns a numerical identifier (ID) to every client that joins and to every game session that is started by the first connected client. The session is ended when the last connected client leaves the game. Each client (`Sender` thread) sends a state update message to the server with the interval of 200 ms or right after a new action is performed, containing the following data:

- **timestamp**: time when the player state snapshot was taken by the client

- **position**: x and z coordinates (y coordinate is fixed and does not need to be transmitted)

- **rotation**: orientation quaternion
- **current action**: idle, walk, punch, handshake, rock-paper-scissors or wave
- **opponent player**: ID of the player participating in the action (if any)

The `SocketReader` thread on the server listens continuously on the socket and receives messages. The `BufferHandler` thread checks every message that is received on each client socket. If the message contains a score-earning action request (punch, handshake or rock-paper-scissors), it is verified by calculating the distance between interacting players and comparing it to the maximum distance allowed to perform the corresponding interaction. If the distance is within the allowed range, the action is acknowledged and the state update message is added to a buffer. Otherwise, if the distance is greater than the maximum allowed distance, the action is discarded and removed from the message. Once all received messages are read and added to the buffer, the contents of the buffer is forwarded to all clients as a state update message.

The `Receiver` thread on the client side dispatches every received message and marks it with a delivery deadline, equal to the sum of current time and the simulated network delay. The `Delay simulation` thread checks every message and sleeps until the time of the deadline before processing it. The data is afterwards delivered to the `Main` thread that performs necessary changes to the player objects.

The entire game simulation, including input processing and executing user commands, as well as moving and rendering the world and player objects, is managed by the `Main` thread on the client side. World objects include some static entities (landscape, trees, buildings, pavements, etc.) and five dynamic entities (four running cats and one flying skeleton). They do not influence the outcome of the game, serving only as background objects. The main components of the client-server architecture, chosen for the test game model, are summarized in Figure 3.15

Every player object is represented by an avatar entity. The state of a local player is controlled by user input, while the states of remote players are reconstructed from the state updates data. By using the control keys (arrow keys for moving and turning, `P, H, R, W` for punch, handshake, rock-paper-scissors, and wave respectively), a player can either move or execute a score-earning action. There are two constrains for moving:

- absence of collision
- no score-earning action currently being executed

If any of these constraints are not satisfied, moving is not possible. Player's ability to execute any of the score-earning actions is restricted by the following require-

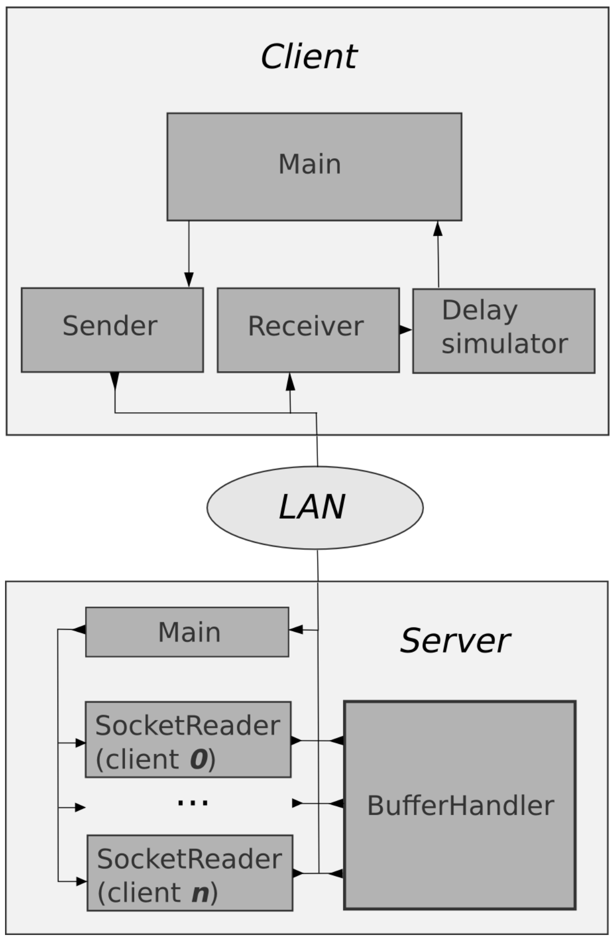


Figure 3.15: Client-server architecture chosen for the test game model

ments:

- presence of at least one remote player within the corresponding action distance range
- no score-earning action currently being executed
- if a score-earning action was executed previously, the player should have moved 500 steps away from the position, where the last action was performed (the information is displayed on the screen for player's convenience)

The following score-earning actions need to be approved by the server: punch, handshake, and rock-paper scissors. These actions require both interacting players to be within the required distance interval. Since player state information on each client may be outdated due to the network delay, inconsistency can occur in clients' decisions on whether an interaction can be performed. It is therefore necessary to let a single authoritative entity make the final decision. Thus, every action is verified by the server after having been performed by a client. A client application is not waiting for the acknowledgement and simply executes the operation, using the last known state of the simulation and assuming that it will be approved. In case the operation is discarded, corresponding player's score is corrected on the client side.

The requirements for the wave action are somewhat different. There is no distance constraint assigned, but both interacting players need to be able to see each other. For the sake of simplicity, we use `Ogre::Camera`[12] object to check if this requirement is fulfilled. To determine whether two players `playerA` and `playerB` can interact by performing a wave action, we call a function implemented in `Ogre::Camera`:

```
bool isVisible (const Vector3 \&vert, FrustumPlane *culledBy=0) const
```

Both participating players can see each other if both of the following function calls return `true`:

```
playerA->playerCam->isVisible(localPlayer->playerNode->getPosition())
playerB->playerCam->isVisible(current->playerNode->getPosition())
```

The server is not replicating the entire simulation state; instead, it makes calculations based on the latest data received from the players. To be able to verify a wave action, the server would need to access the camera objects of the corresponding players, which is not possible. We can solve this problem in two different ways:

- by setting up OGRE framework on the server and let the server access desired objects, when necessary

[12]http://www.ogre3d.org/docs/api/html/classOgre_1_1Camera.html

- by letting the client itself verify the wave action

Considering that the only application of simulating the game world on the server would be the ability to verify wave action, this solution could introduce much unnecessary overhead. Therefore, we decide to let the action be approved on the client side. To achieve that, we implement a method that allows a client application to bypass the simulated delay and access the updated state information, when verifying a wave:

```
bool Game::canWave(Player* player)
{
   Vector3 prevPos = player->playerNode->_getDerivedPosition();
   Quaternion prevOrient =
      player->playerNode->_getDerivedOrientation();

   // temporarily change the player state to what is currently on
      the server
   player->playerNode->setPosition(player->realPos);
   player->playerNode->setOrientation(player->realOrient);

   bool canwave = false;

   Vector3 myPosition = localPlayer->playerNode->getPosition();

   // check if players can see each other
   if (player->playerCam->isVisible(myPosition) &&
      localPlayer->playerCam->isVisible(player->realPos))
      canwave = true;

   // restore the state to what is currently seen by the client
   player->playerNode->setPosition(prevPos);
   player->playerNode->setOrientation(prevOrient);

   return canwave;
}
```

Each time a state update message is received, the `Receiver` thread stores the corresponding remote player's position and rotation information in `player->realPos` and `player->realOrient` respectively. This data is only accessed by the `Main` thread when verifying a wave action initiated by the local player. In this way we simulate a verification procedure equivalent to the server's acknowledgements for other types of actions.

3.3.2 Entity Interpolation

Any motion in a virtual world is performed by shifting the positions of movable objects and producing an image that reflects how the world looks like. Typically, a virtual camera defines from which perspective the "picture" of the world is taken. That image is then being rendered to the screen and is usually referred to as a **frame**. Normally, to display motion that a human eye would perceive as smooth and realistic, at least 24 frames should be rendered per second. However, that is not the only requirement. In contrast to reflecting real world motion on a video, where motion happens naturally, in virtual world, we also simulate the motion itself.

Rendering engines, like OGRE, are usually based on a frame event loop, where necessary calculations and changes to the world objects are being done in between rendering the frames. An OGRE application should first go through the initialization process, which is well described by Junker (2006) in Chapter 4. A simple way to manage the frame rendering loop in OGRE is creating a subclass of `Ogre::Framelistener` class and overriding one or more of its following functions:

- `virtual bool frameStarted (const FrameEvent &evt)` - called when a frame is about to start rendering;

- `virtual bool frameRenderingQueued (const FrameEvent &evt)` - called after all render targets have issued their rendering commands, but before the rendering buffers get flipped over;

- `virtual bool frameEnded (const FrameEvent &evt)` - called right after a frame is rendered.

The frame rendering loop in our test game implementation is managed by a class that inherits `Ogre::FrameListener`. We do all the necessary changes to the movable objects in the game and process user input in `frameRenderingQueued` function:

```
bool Game::frameRenderingQueued(const FrameEvent &evt)
{
   moveWorldObjects(evt);
   printStepsLeft();
   printPlayerScore();
   printTime();

   if(!processUnbufferedInput(evt))
      return false;

   managePlayerStates(evt);

   return BaseApplication::frameRenderingQueued(evt);
}
```

By updating motion states of the objects each time a new frame is queued for rendering and taking into account the time that passed since last frame was rendered in calculating new positions of the objects, we simulate motion that is smooth. While the local player entity is being controlled by user input, all the remote player entities are not. The information about their kinematic states is extracted from the state update messages, which are generally received with the interval of 200 ms. If we updated the positions of remote players only upon an update message arrival, the motion would no longer look realistic and the user would see remote players jumping from one position to another. Doing that should be avoided, therefore, remote players' motion needs to be evenly interpolated between the known positions.

For interpolating between avatar's positions in the virtual world space, we have considered two different solutions:

- **linear interpolation**, where the interpolant is a linear polynomial.
- **spline interpolation**, where the interpolant is a spline (special type of piecewise polynomial)

To perform linear interpolation, we draw a straight line between the points, and then, gradually move the object along the obtained path; for spline interpolation, we draw a curve through the points. Connecting two points with a straight line is rather simple and does not need detailed explanation. However, spline interpolation is more complex. Firstly, there are different types of splines. **Cubic spline** and **Cubic Bezier curve** (Figure 3.16 [13]) are quite often used for calculating motion paths in computer graphics and related fields. Another representative of the cubic splines family is **Catmull-Rom spline** (Figure 3.17[14]). In a Catmull-Rom spline, the tangent at each point is calculated by using the previous and the next point on the spline, which causes the curvature to vary linearly over the curve segment's length. (Catmull et al., 1974) Secondly, we typically need at least three points to build a cubic spline.

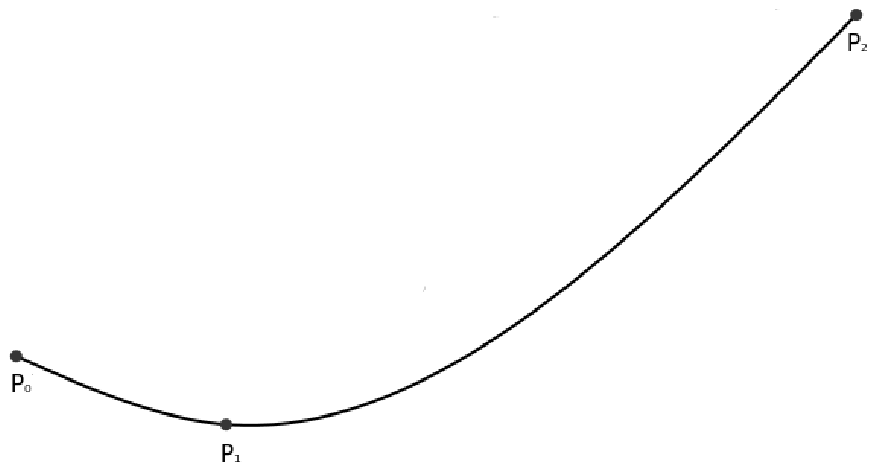
Whether linear or spline interpolation is a better choice generally depends on the interval length between the points and, of course, on the physical characteristics of the object's motion. For a vehicle object, spline interpolation might be a better choice, since the path of a vehicle's motion is usually a curve. For an avatar, it depends on the characterictic features of its motion.

The first solution alternative we tried was `Ogre::SimpleSpline`[15] that implements a Catmull-Rom spline. We used three control points to create a spline:

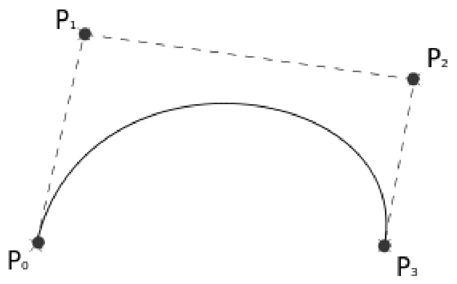
[13]The figure is inspired by Wikimedia Commons files
http://upload.wikimedia.org/wikipedia/commons/e/e2/Cubic_splines_three_points.svg
http://upload.wikimedia.org/wikipedia/commons/d/d0/Bezier_curve.svg

[14]The figure is inspired by the "Introduction to Catmull-Rom Splines" article http://www.mvps.org/directx/articles/catmull/

[15]http://www.ogre3d.org/docs/api/html/classOgre_1_1SimpleSpline.html



(a) Cubic "natural" spline



(b) Cubic Bezier curve

Figure 3.16: Common splines used in computer graphics

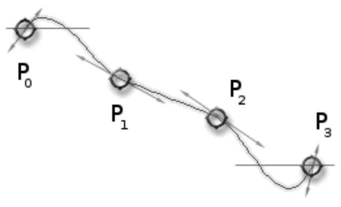


Figure 3.17: Catmull-Rom spline

- 1: avatar's previous position, received as position update before it moved to the current position (if last received update is u_n, then previous position is u_{n-2})
- 2: avatar's current position
- 3: last received position update (destination)

In addition, we used the option of calculating the tangents automatically, when a new point is added to the spline. As a result, in some cases, the interpolated motion path contained excessive curvature; thereby, avatar's motion was not smooth and somewhat tottering. Troubleshooting in this case appeared to be not that straightforward for several reasons:

- It is not possible to access tangents at control points or along the spline's length. Therefore, we could not determine whether the problem occured due to miscalculated tangents.
- `SimpleSpline::interpolate(Real t)` does not seem to produce accurate interpolation, if the control points are not spaced evenly.
- There is no way to check the spline's length.

Finally, we decided that such interpolation type was not optimal for our prototype.

Thus, we conluded that for a fine-grained motion path of an avatar, linear interpolation can be more effective, provided that the interval between the points is short enough, like in our case. Therefore, linear interpolation was our final choice, since it produced the desired smooth motion, while the outcome was stable. The algorithm we implemented is rather simple:

```
// player's current position
Vector3 currentPos = player->playerNode->_getDerivedPosition();
// player's last known position, received as an update
Vector3 destinationPos = player->positionUpdate;

// the distance that an avatar would have covered since last frame
Real perFrameDistance = SPEED_PER_SECOND*evt.timeSinceLastFrame;

// total distance to the destination
Real totalDistance = distanceToPosition(destinationPos, currentPos);

// for smooth motion, only move the avatar if the total distance is
   significant enough
if (totalDistance >= perFrameDistance)
{
   Vector3 direction = destinationPos.operator-(currentPos);
   direction.normalise();

   player->playerNode->translate(direction*perFrameDistance,
      Node::TS_WORLD);
}
```

Position interpolation only is not enough to make an avatar's motion look smooth and realistic. It is important to make sure that rotations are handled correctly. There are different alternatives that can be used as a solution for that. Again, the choice would generally depend on the avatar's motion properties. Since we use linear position interpolation, it is convenient to interpolate between rotations linearly as well. Rotation interpolation is implemented as follows:

```
Real perFrameRotation = evt.timeSinceLastFrame;
Quaternion nextOrient = player->orientationUpdate;
Radian totalRotation = nextOrient.getYaw();
Quaternion curOrient = player->playerNode->_getDerivedOrientation();
Radian curRotation = curOrient.getYaw();

// difference between current rotation and destination rotation in
   radians
Real rot =
   Math::Abs((totalRotation.operator-(curRotation)).valueRadians());
// rotation per frame - converted from degrees to radians
Real rotPerFrame = (DEGREES_PER_SECOND* 0.0174532925)*
   evt.timeSinceLastFrame;
Real rotAngle = rotPerFrame / rot;

if (rot > 0)
{
   if (rotAngle < 1 && rotAngle > rotPerFrame)
   {
      Quaternion dest = Quaternion::nlerp(rotAngle, curOrient,
         nextOrient, false);
      player->playerNode->setOrientation(dest);
   }
   // the difference is too small for interpolation
   else
   {
      player->playerNode->setOrientation(nextOrient);
      player->orientInterpolation = 0;
   }
}
```

When using any kind of interpolation, it is important to remember that interpolating between known positions of a remote player causes certain delay in replicating its motion path. Firstly, it takes time for the update message to be delivered. Secondly, we do not immediately place the player to the last known position; instead, the object is moved there gradually, which causes additional delay. This means that remote player's position will normally deviate from its correct position. In case player's position itself does not significantly influence the game outcome, such deviation can be acceptable. However, it still leads to certain inconsistencies in presentation of the player's state on different clients. Most video games use client side prediction and latency compensating techniques to solve the problem.

3.3.3 Client Side Prediction and Latency Compensation

An important factor in most distributed virtual environments is providing consistency. The concept of consistency has received much attention in both theory and practice. Mostly, the work done in this area focused on applications that are discrete and change their states only according to user input. Examples of such applications are shared text editors or drawing tools. However, some distributed environments, such as computer games and virtual reality simulators, are continuous, which means that they do not only change their state in response to user operations, but also, with passing time. For this category of applications, the issue of consistency is still not clearly defined. Since approaches that are usually applied to achieve consistency in discrete applications ignore the state changes caused by passing time, they will not give the same result when used in continuous applications. At the same time, the issue of consistency itself can be categorized as short-term consistency and long-term consistency. Whether the goal is maintaining short-term or long-term consistency depends on the characteristics of the application as well. (Mauve et al., 2004)

In the context of distributed networked virtual environments, such as games or simulation applications, we refer to consistency as maintaining correct kinematic states of all entities on every client. Such consistency is often difficult to achieve due to the network transmission delay, since latency may lead to incorrect state of distributed objects on remote clients. The problem is usually approached by using latency compensating techniques on the client side. There is, unfortunately, no universal method to achieve consistency for any environment, neither it is always possible to provide full consistency at all times. A typical solution is attempting to reduce the deviations as much as possible, and eventually, correct the situation. The choice of the right technique for that depends on the type and properties of the environment itself, while, in almost any case, there is a tradeoff between consistency and application's responsiveness.

Dead Reckoning

The most common client side prediction method is **dead reckoning**. Dead reckoning combines state prediction and state transmission, and does not necessarily require a centralized server. The main principle of this technique is predicting the current state of an entity according to its last known state. Examples of such entities are vehicles, such as a car or a plane, avatars or bullets. The parameters used in predicting new state of an entity include last known position, direction, and velocity. Errors in computations are likely in almost any case, since remote entities are usually controlled by their users. The accuracy of dead reckoning is considerably dependent on how much the physics laws influence entity's motion behavior. The maximum possible threshold in difference between actual and predicted state depends on how fast the state updates are being delivered in correlation to how fast the changes in entity's state can take place. For example, predicting new state of an airplane would be simpler than estimating new state of an avatar. An airplane can take off, accelerate, slow down, fall or land. However, it cannot suddenly stop and hang in the air, so, changes in airplane's velocity, as well as direction, cannot exceed certain limits, which narrows down the range of possible changes in its state.

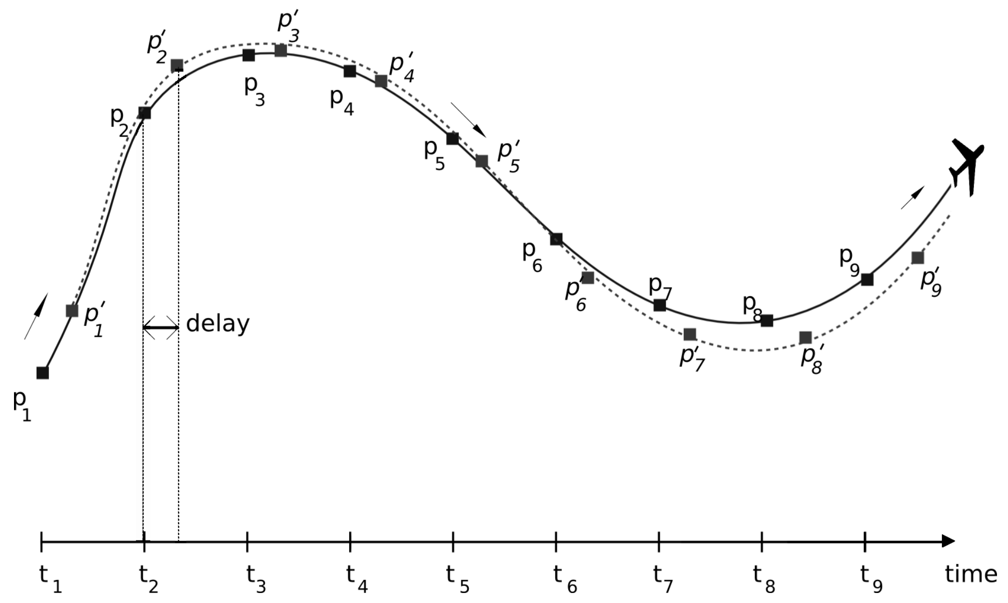


Figure 3.18: Predicting the motion path of an airplane by using dead reckoning

Consider a simple example shown on Figure 3.18, where we illustate an approximation of calculating the motion path of an airplane from the point t_1 to the point t_9 on the timeline. The solid line is the object's actual path, while the dotted line is the predicted path. State updates are sent by the client application A that is steering the object and are denoted as p_n. Each update is received by another client application (B) with some delay and is used to extrapolate the object's current position (p'_n). When B receives the first update p_1, it extrapolates the airplane's position at the current time (t'_1) and continues the extrapolation until next update(p_2) is received. At that moment, the airplane's direction has changed slightly, so the extrapolation continues from the current object's position (p'_2) to the next predicted position (p'_3). The same strategy is applied each time a new state update has arrived. As we can see, the predicted path deviates slightly from the actual path, but the deviation is not drastic, since the error rate is reduced by the restrictions of the environment's physical model. Therefore, it can be suggested that using dead reckoning for prediction of a smooth motion (such as motion of a vehicle) is rather effective.

Pantel and Wolf (2002b) have studied the suitability of dead reckoning schemes for games, using sports, 3D-action, and racing games for investigation. The authors have considered various position and input prediction schemes, as well as a scheme with no prediction. Measurable differences were noticed in the results produced by different schemes, where the most complex schemes did not always give the best results. The authors pointed out that the choice of prediction scheme was significantly dependent on the game type. Their measurements showed that input prediction was rather successful in racing games, where the player objects were represented by vehicles. However, the authors figured out that prediction of players' movements in sports games, where the motions were not as smooth and dependent on the physics as in the vehicle-like movement model, was not so easy to handle. Therefore, using prediction schemes was concluded to be ineffective for such game models, and presentation delay was suggested to be used instead. Nevertheless, it was mentioned that the proposed method would introduce an additional delay and could only be used in game models that would tolerate it.

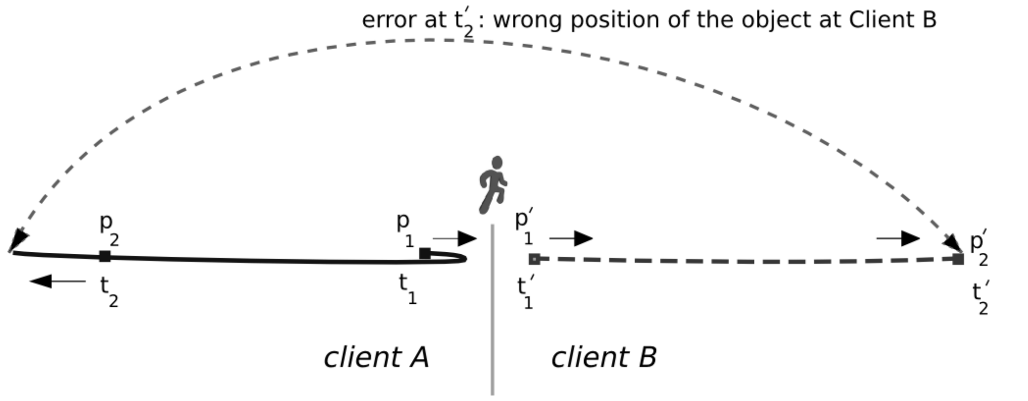


Figure 3.19: Predicting the motion path of an avatar by using dead reckoning

The situation is indeed different when it comes to prediction of motion that is not significantly affected by intertia or other physical constraints, like motion of an avatar. Consider an example of an avatar that simulates a human being in a distributed virtual environment, where several clients replicate its actions, receiving updates about its kinematic state with certain frequency. Each update is sent by the client controlling avatar's actions at the time t_n, with an interval of $t_n - t_{n-1}$, and is delivered to the other clients with some latency l at the time t'_n, which is calculated as follows:

$$t'_n = t_n + l$$

The avatar can move with the velocity v, change its direction d or remain motionless. In contrast to the airplane object discussed previously, the avatar can stop moving any time and change its direction unexpectedly. For the sake of simplicity, we assume that there is no acceleration in the object's velocity, so, it either moves with constant velocity or remains motionless. Figure 3.19 shows possible development of the avatar's kinematic state over time. The solid line represents its actual state, while the dashed line is the state predicted by using dead reckoning. Dead reckoning assumes that the object will continue moving in the same direction. Therefore, if we receive a state update, indicating that at the point t_1 in time the avatar was at position p_1, moving in direction d with constant velocity v, we assume that by current time, the avatar has moved to position p'_1, which is calculated as follows:

$$p'_1 = p_1 + (d * v * (t'_1 - t_1)$$

Using the same method, we continue the extrapolation until the next state update is received, moving the avatar to position p'_2. However, once the state update is received at t'_2, we figure out that instead of moving in the same direction, as we expected, the avatar had turned around and moved in the opposite direction. The maximum possible error in the calculation of avatar's position at t'_2 (the distance between the actual and predicted positions) is therefore the maximun distance that can be covered by the avatar in a time interval equal to the sum of the time between two updates and delay time multiplied by two, since the avatar could have moved in the opposite direction:

$$error \sim v * (t_n - t_{n-1} + l) * 2$$

Efficiently correcting the error by interpolating between the avatar's current position and its real position is not possible if the speed is constant - the distance that we need to move the avatar to within certain time limit is then greater than what the avatar is able to move to. It might also increase the maximum error in the long run. On the other side, forcefully placing the avatar to the desired position cannot be left unnoticeable to the user and would no longer provide a smooth user experience.

If we do not use any prediction and simply interpolate between the positions that we get in each state update message, the path of the avatar's motion does not change, but rendering of all the movements is delayed by the latency time plus interpolation time. Nevertheless, in this case, the maximum difference between the avatar's real position and its current position, as presented at the remote client replicating the avatar's actions, at the time t'_n is constant and equal to the distance that the avatar can cover in the time it takes to interpolate between avatar's current position and the latest state update (the time interval between two updates), plus the latency time:

$$error \sim v * (t_n - t_{n-1} + l)$$

Since the maximum possible error in position estimations by using dead reckoning in such case is about twice as large, comparing to using no prediction at all, dead reckoning appears to be wrong choice for a virtual environment model where player objects behave in a similar way.

AntReckoning

Obviously, prediction of avatar's actions based on last known state only can be rather ineffective, especially if there is little or no influence of physics on the motion. Prediction, if used, should therefore take other factors into account. Generally, the behavior of players in a game is influenced by their goals and interests. Considering player's goals and interests is a key aspect in **AntReckoning** - an interest-based approach to dead reckoning introduced by Yahyavi et al. (2011). AntReckoning is a dead reckoning algorithm inspired by the behavior of ant colonies. The algorithm is based on incorporating players' interests and goals into dead reckoning calculations. *Pheromones*, or *attraction forces*, are used to model such interests, considering also temporal and spatial factors. Each entity in the game is assigned a certain level of attractiveness, which leads to spreading pheromones in the game world. The game world is in turn divided into non-overlapping segments called *cells*. The attractiveness of each particular cell influences the dead reckoning decision of whether the player will be interested in moving towards or away from it. For example, a player might want to approach its opponent when attacking or move away when being attacked. Pheromones are also generated by other objects that might be of some interest to the player, such as health packages. The attraction forces are spread in the game world, which means that their concentrations in neighboring cells are mutually dependent. Pheromones fade over time, which avoids infinite accumulation of them in the game world.

Yahyavi et al. (2011) evaluated AntReckoning both in comparison to traditional dead reckoning, to estimate any potential gains, and independently, by performing the sensitivity analysis. Traces collected from Quake III and World of Warcraft were used for evaluation. The methodology was based on dividing time in frames, setting players' positions for every frame as last position updates in the matching time interval, and then, performing both traditional dead reckoning and AntReckoning to figure out which algorithm would give better results. Since the actual positions were known, the performance of each method was evaluated by comparing the calculated distance between the estimated and the actual positions. The basic version of AntReckoning was used, where only players generated pheromones, and equal amount of pheromones was generated by each player, regardless of the state; dissemination of the pheromones was not considered, meaning that only the cells that players went through contained pheromones. The following factors were considered in the sensitivity analysis: the diameter of the region of attraction, the attractiveness of players to each other, the evaporation factor, and the duration of the prediction step.

The results of the experiments formed a rather interesting outcome. In general, AntReckoning introduced an up to 30% improvement in prediction accuracy over traditional dead reckoning, according to the results on Quake III and World of Warcraft. Nevertheless, the sensitivity evaluation showed that the outcome was dependent on the suitability of values chosen for certain parameters. Increasing the size of the attraction region and taking larger number of attraction forces into account improved the quality of predictions only within certain size limit. Increasing the size beyond that limit appeared to be counterproductive and affected the performance negatively. Similarly, considering attraction forces to certain extent improved the accuracy of prediction, while overestimating those resulted in decreased efficiency. Evaporation appeared to have insignificant effect in Quake III, which seemed to be caused by the fast pace of the game, while in slower-paced World of Warcraft where players were motivated by long-term goals, the evaporation factor had a higher impact: slower evaporation improved the performance.

It was stated that AntReckoning had no effect on predicting human interactions, since they were an order of magnitude slower than game events. For this reason, it was proposed to only use inertia for predicting avatars' movements in short periods of time (such as 50 ms). AntReckoning performed better in long-term prediction, where players' goals further in the future (up to seconds) were considered. All in all, AntReckoning appeared to be an effective prediction algorithm only if it was fine tuned to a particular game model and if players' long-term objectives could be considered, while failures in tuning could even lead to negative influence on the game outcome. (Yahyavi et al., 2011)

Local-Lag

Local-lag is a method based on voluntarily decreasing the application's responsiveness to eliminate short-term inconsistencies. The method delays an operation by certain amount of time instead of executing it immediately. Consider an example where a user U_1 performs an operation that is replicated by a user U_2. The current time is t^0, and the operation is due for execution at t^*, provided that $t^* > t^0$. If the difference between t^* and t^0 is large enough for U_2 to be able to receive the operation before t^*, both U_1 and U_2 would execute the operation at the right moment in time and eliminate short-term inconsistencies. On the other hand, even though inconsistency is prevented, the operation is not executed immediately at U_2, which may be noticeable or even distracting for the user. Therefore, local lag is only effective if the sufficient value is assigned to time interval by which all operations are delayed. It should be large enough for an operation to be delivered on time, but also, short enough to prevent user dissatisfaction. (Mauve et al., 2004)

Local lag is used in many networked computer games. However, the tradeoff between responsiveness and eliminating inconsistencies is not always successfully achieved. It is often hard to estimate correct operation delay time to be used by local lag, since network delays can vary, depending, for example, on the player location. For the purpose of keeping the local lag delay as low as possible but still sufficient, generally, no constant delay interval is used. Instead, it is defined dynamically by the operation delivery acknowledgement, meaning that user operation is executed only after it has been approved, which, in some cases, decreases the playability of the game significantly.

Timewarp

While client-side prediction methods and local-lag are used to prevent possible inconsistencies, they still do not guarantee error-free states of replicated objects. **Timewarp** is a method of repairing the states after an inconsistency has occurred. It requires that an application instance i maintains a list of performed operations L_i^o and a list of states L_i^s. Each application instance is initialized at a certain point in time t_i^I with the correct state $s_{i,t_i{}^I} = s_{P,t_i^I}$. It is assumed that the lists are sorted by t^* and t. At initialization, L_i^o is set to be empty and L_i^s is set to only contain $s_{i,t_i{}^I}$. With these preconditions, Mauve et al. (2004) describe the timewarp algorithm as follows:

- *Step 1.* Define T as constant frequency of calculating and presenting new state to the user. Wait for T, receiving local and remote operations and storing them in L_i^o. Mark ${t_w}^o$ as the smallest t^* of any operation that was performed during T.

- *Step 2.* Find the state that directly precedes the earliest operation received in step 1, denote the time when this state was valid as t_w^s.

- *Step 3.* Apply all the operations to the state determined in step 2 that happened after the time that state was valid (${t_w}^s$) in a fast-forward mode. To do that, determine the state before each operation should have been performed and then, execute it; replace the states in the list by updated versions that consider the events arrived during step 1.

- *Step 4.* Provided that t^c is the current time, use the state obtained at step 3 to calculate the state at t^c, save it to the list of states L_i^s, and display to the user.

Discussion

The client-side prediction, latency compensation, and state correction methods aim for the same target - eliminating or reducing the effect of network transmission delay that usually leads to inconsistencies. The choice of the right technique to solve the problem depends on the characteristics of the virtual environment model. In our test game model, we do not implement any of the techniques discussed above. We cannot use dead reckoning effectively, because the physical model of the game has too little influence on avatar's motion. AntReckoning requires rather clear definition of the attraction forces, which cannot be applied to our game model. Local lag could be used in cases when latency is relatively small, but it would unacceptably decrease the game's responsiveness at higher latencies. Therefore, we only use entity interpolation on the client side. Nevertheless, since the players' positions are key determinants in deciding whether an interaction is possible, we use players' last known (instead of interpolated) positions in calculating the distance between them when an interaction is requested. Still, it does not prevent the client from making wrong decisions, because the last known position can be outdated due to the delay in updates delivery.

In our test implementation, we only have two factors that reflect the simulation status of a player entity (in this context, we do not refer to player's kinematic state, but rather to the characteristics that determine player's progress in the game):

- objective status that is represented by the player's score only
- user's subjective perception of the player's status, according to the operations that were seen to be executed

To repair the simulation state in case of an error on the client side, we use a simple correction method that can be described as follows:

- *Step 1*. Process user input at current time t^c and determine whether the currently requested operation o^c is legal, according to the current client state. If the operation is legal, save the current state of the local player in s^c. Execute the operation, making necessary changes to the states of involved players.
- *Step 2*. Send a state update s^c immediately and let the server make the final decision on whether the operation o^c was legal t^c.
- *Step 3*. If a score correction request is received from the server, it means the operation was illegal. To correct the state, replace player's score with the value received from the server.

Ideally, on the server side, we need to use a mechanism that keeps a list of operations performed during a certain period of time in the past, starting from current time (similar to timewarp). Assuming that any client's operation request arrives to the server with some delay, we denote the time of delivery as t^d and the corresponding state of simulation as s^d. To decide whether an operation o^c was legal at the time t^c, the server would need to revert the state of the whole simulation to the state s^c that corresponds t^c, and make a decision afterwards. In our client-server prototype, the network latency is only simulated on the client side upon message reception, but never when a client sends a message to the server. Since we perform experiments in a LAN where the network delay is smaller than 1 ms, we assume that the difference between t^c and t^d is insignificant. Therefore, we expect that the difference between s^d and s^c is neither considerable and suppose that $s^d \sim s^c$. Thus, the server can use s^d to verify the operation o^c.

The method described above corrects inconsistencies in the player's score, but it does not influence player's nonobjective observations. Once an operation is executed and seen by the user, there is obviously no way to correct the user's subjective perception of the player's status.

3.4 Summary

In this chapter, we went through the details of game model design and implementation. We argumented for the choice of human avatar design and animation tools, game development tools, physics simulation method, and explained the process of integrating each separate component into the prototype that we use for experiments. Also, we described the categories of player interactions and client-server architecture, as well as alternatives for client-side prediction and latency compensation techniques. In the next chapter, we discuss our experiments and evaluate the results.

Chapter 4

Evaluation and Discussion

In this chapter, we present our hypothesis, describe the experiments constructed to verify it, and analyze the results. The chapter is organized as follows. In the first section, we state our expectations to the outcome of the experiments; in the second section, we describe the details of the experiments; in the third section, we analyze the results; finally, we describe observed user behavior and feedback.

4.1 Hypothesis

Related work that focused on the latency sensitivity of different player interaction types suggests that fast-paced interactions, such as, for example, precision shooting, are more sensitive to network delay than events that do not require immediate response, such as building or exploration. To the best of our knowledge, and as already mentioned in previous chapters, no research has been done to establish whether the distance range is a determinant factor in estimating latency sensitivity of a particular interaction category.

We hypothesize that distance range plays an important role in determining latency sensitivity of a player interaction. The latency sensitivity is defined by the influence of delay on player's performance, namely, player's ability to fulfill corresponding interaction. Latency-sensitive actions are expected to be much harder to perform at increased levels of latency, meaning that if player performance depends on such interactions, it degrades when the network delay increases. In contrast, we expect that the possibility of performing latency-insensitive interactions is not significantly affected by increased levels of latency, provided that the increase is within reasonable limits.

Both short- and long-range interactions are implemented in our prototype, each category being represented by two different actions. Since the distance range varies even for actions within same category, each of them is assigned a range index for the convenience of evaluation. The indices are listed in Table 4.1

Table 4.1: Distance range indices

Category	Action	Range index
Short-range	Punch	**1.5**
	Handshake	**2.5**
Long-range	Rock-paper-scissors	**10**
	Wave	**inf**

Each range index is derived from the distance range constraint applied to the corresponding action. Short-range interactions have indices below 2.5, while long-range interactions have indices over 10. We set a range distance of the wave action to infinite, since there is no distance constraint assigned to it.

We expect the latency sensitivity of each action to be inversely proportional to its range index (the higher the index, the lower the latency sensitivity). Therefore, we believe that short-range interactions will not be able to tolerate a significant increase in latency, in contrast to long-range interactions, which are expected to tolerate large delays (within reasonable limit). Inspired by latency sensitivity findings in literature (discussed in Chapter 2), we establish expected tolerable latency threshold values (listed in Table 4.2).

Table 4.2: Expected tolerable latency thresholds

Category	Action	Latency threshold
Short-range	Punch	**100-200 ms**
	Handshake	**200-300 ms**
Long-range	Rock-paper-scissors	**500 ms**
	Wave	**1000 ms**

Since scoring each action a certain amount of times is a requirement to win, we believe that players' winning probability will be negatively affected by increased latency. Therefore, we expect that the total number of winners among all participants will become smaller at higher delays.

Our hypothesis can be summed up by the following expectations:

- short-range player interactions are more latency sensitive than long-range interactions
- winning a game becomes harder as the latency increases
- completing a game round takes longer as the latency increases

4.2 Experiments

To verify our hypothesis, we performed a series of experiments with 22 participants. We did not take participants' age, gender, skills, or gaming experience into account. Nevertheless, all the participants were unfamiliar with our game model, therefore, we assume that users' skills, gaming experience or similar factors could generally not influence the outcome of the tests.

We used 7 Ubuntu machines, each of them had at least 2 GB of RAM; 6 machines were running client applications, and 1 machine was running the server on three different ports (in case we needed to use all the client machines at the same time). All the tests were performed on a LAN with latency less than 1 ms. Each experiment involved two participants playing against each other and consisted of 6 rounds. During each round, we simulated various network delays, as listed in Table 4.3.

Table 4.3: Test rounds sequence

Round number	**Simulated latency (ms)**
1	0
2	100
3	200
4	300
5	500
6	1000

Generally, the simulated values of latency would need to be shuffled throughout the rounds sequence. That is usually done to prevent the users from guessing what and how is being simulated, and also, to avoid giving them an impression that the situation is getting worse. However, our case is somewhat different from a typical case. We do not use a game that the participants have played before. Therefore, we want the testers to become familiar with the game model in the beginning. To let them get an appropriate impression of the game, we do not want to simulate any delay during the first round. If we simulate large delay already in the second round, for example, the testers may get unmotivated too early. Therefore, we believe that increasing the delay value gradually is a suitable solution for our experiments. Like that, the users will most likely feel some difference during last rounds, and that is something that we want to investigate. If we shuffle the simulated delay values, we risk to prevent the players from getting any subjective impression, except confusion.

Before we started the experiments, all the participants were informed about the rules of the game and encouraged to win. Winning requirements were presented to the players as follows:

- score 10 punches, 10 handshakes, 10 rock-paper-scissors, and 10 waves
- fulfill the score requirements within the time limit of 350 seconds

The players were not informed of the delay simulation and were only asked to play 6 separate rounds, trying to fulfill the requirements. During every round, we logged the number of player's attempts to perform each type of action and the number of times the corresponding action type was acknowledged by the server. For example, if a player attempted to perform 10 punches, but 4 of them were discarded by the server, the number of punches in the final score statistics was logged as 10:4.

Initially, we intended to use both objective and subjective measurements for player performance evaluation, namely, objective score statistics and subjective MOS ratings (discussed in Chapter 2). To obtain MOS, we planned to ask each player to rate their perception of the gameplay quality during each round by giving the corresponding score value. Nevertheless, it turned out that many players considered it difficult to provide such subjective ratings. We believe that possible reasons for that included players being unfamiliar with the game and uncertain regarding which factors should be considered while giving the evaluation. Therefore, we decided to omit asking players to provide subjective ratings and observed their behavior during each round instead.

Though all players were encouraged to win, we expected that only some percentage of the participants would be able to achieve that. Therefore, we logged every player's success or failure to win and total duration of the round in seconds.

4.3 Results

The results of the experiments supported most of our expectations. Figure 4.3 shows a median score statistics per player for every round. The short-range player interactions are represented by the shades of red, while the long-range interactions are expressed by the shades of blue. Generally, we noticed that the score earned for performing short-range interactions (punch and handshake) had a tendency to decrease as the latency increased. It means that most players managed to score more punches and handshakes while the network delay was low or zero, but when we introduced more latency, scoring a punch or a handshake appeared to be much harder. In contrast, the long-range interactions score remained relatively stable, regardless of increase in latency. The reason is that scoring a long-range action at higher network delay was still feasible.

Delayed delivery of updates enabled a client application to use outdated positions in making decisions. The higher the delay, the greater the possibility of wrong decision on the client side. All wrong decisions were eventually corrected by the server; as a result, the players scored less actions than they attempted to perform. Figure 4.2 illustrates the influence of latency on the amount of actions discarded by the server (the statistics is represented by a median result for all participating players).

According to the outcome of the tests, the percentage of discarded punch attempts increased proportionally to the increase in delay, starting from latency of 100 ms (as expected). There was a similar trend with handshakes, but it started at 500 ms, slightly later than we expected. Long-range interactions appeared to be rather resistant to latency, though when the delay reached 500 ms, a small percentage of rock-paper-scissors attempts was also discarded. Since the degradation was insignificant at the delay of 500 ms, we assume that rock-paper-scissors action would be able to tolerate latencies of up to $\sim$400 ms. Waves remained rather tolerable to different levels of latency throughout the entire set of tests, keeping a median result of 100% of all action attempts approved by the server. We did not use higher delay than 1000 ms in our experiments, and all the estimations were done in regard to the prototype we have implemented. Since the participants were able to perform any type of action, the latency ranges had to be adjusted to fit the game model on the whole. Therefore, we considered that simulating delays above 1000 ms would no longer keep the game playable. As 1000 is the highest latency we tested, we conclude that the tolerable delay for the wave action is at least 1000 ms.

As a result of the observed trend in players' score statistics, the winning probability appeared to be dependent on the simulated latency. During the entire series of experiments with all participants, there were 66 game rounds in total (11 for each level of simulated latency). Among those, only 17 were won, meaning that one of the participating players in those rounds succeeded in fulfilling the requirements. The winning statistics over all performed experiments is shown on Figure 4.3. The largest amount of the rounds that were won (5) had zero simulated latency; 4 rounds were won in each set of experiments with 200, 300, and 500 ms of simulated latency respectively;

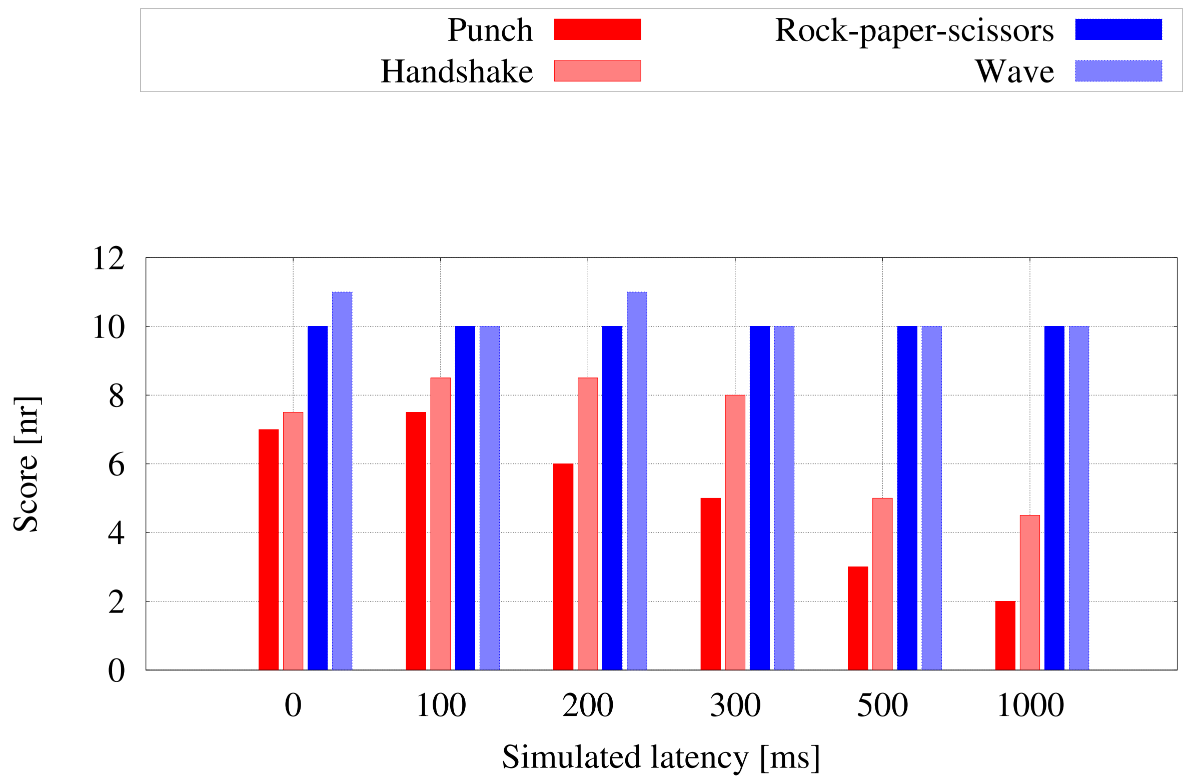


Figure 4.1: Median score statistics per round

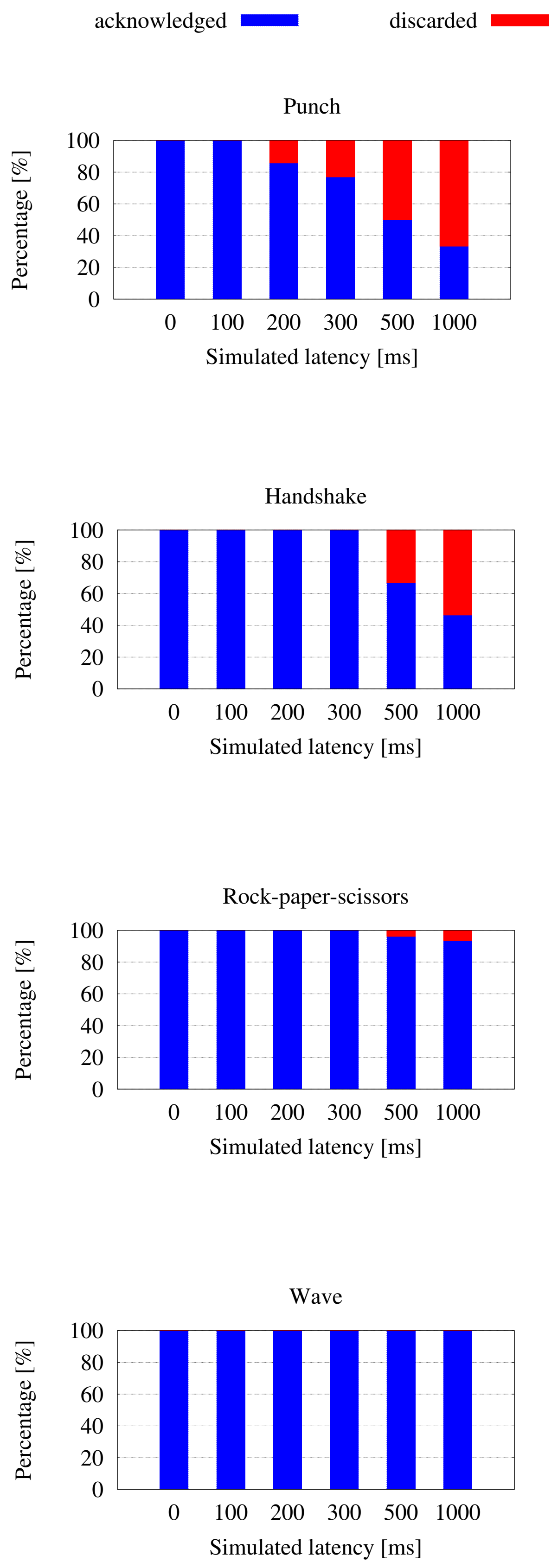


Figure 4.2: Acknowledged vs. discarded ratio

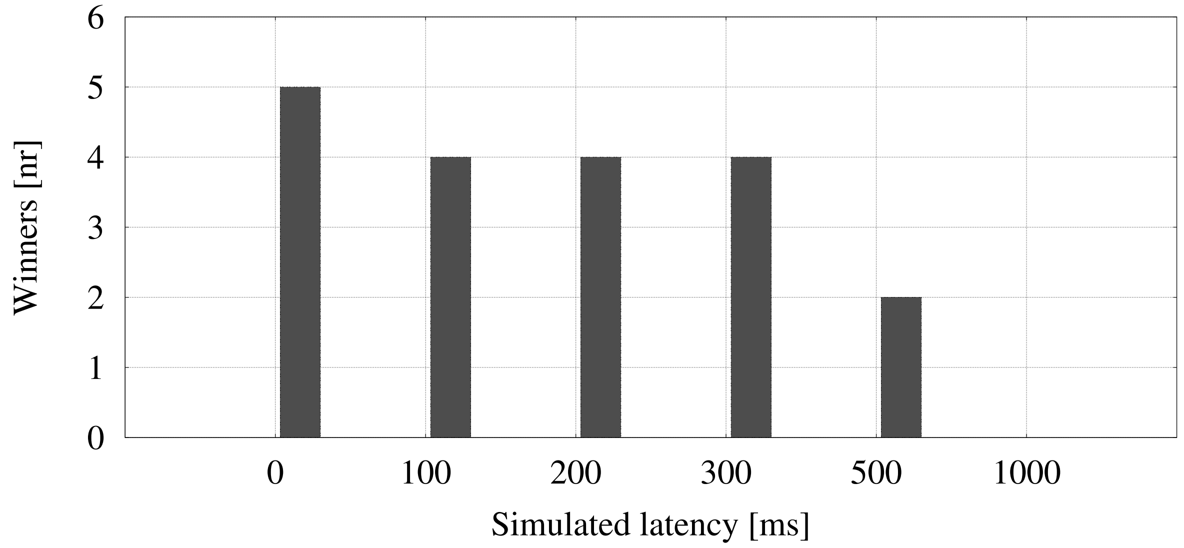


Figure 4.3: Total number of winners among all the participants for all test rounds

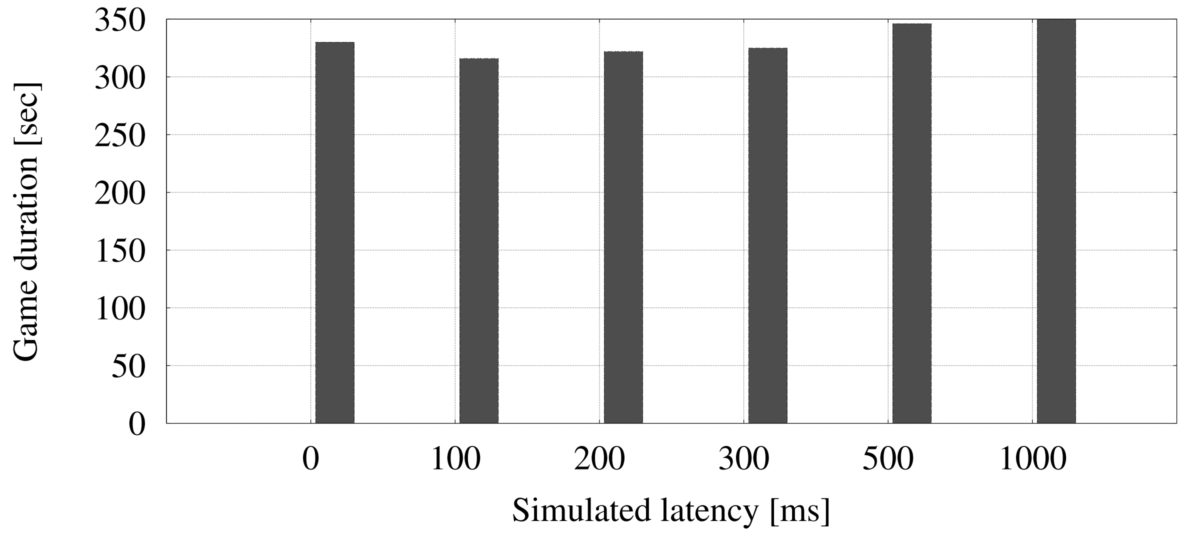


Figure 4.4: Average duration of a single round

nevertheless, no players managed to fulfill the winning requirements when latency reached one second. Such outcome has to some extent reflected our hypothesis, since the overall winning performance was best when zero latency was simulated and worst at maximum simulated latency. However, the trend was not linear, while the results seemed to be somewhat undetermined.

The duration of each round was limited to 350 seconds, meaning that even if players needed more time to fulfill the requirements, they would not be able to get that. At the same time, according to our expectations, a player would, on average, need at least 200-250 seconds to achieve the required score. Therefore, the duration of every round was generally between 200 and 350 seconds. The average round duration for each level of simulated latency is shown on Figure 4.3. There was a slight linear increase in the round duration, starting from latency of 100 ms. That points out that players needed more time to score the required amount of actions when the delay was higher, which we consider to be a degradation in performance.

It is interesting to notice that player performance in general was slightly worse in the first round, when zero latency was simulated, than in the second round, with additional 100 ms of latency. We can observe this trend in the median score statistics per player on Figure 4.3: there were somewhat less punches and handshakes scored in the first round than in the second. Also, on Figure 4.3, we see that the average duration of first round was to some degree longer than that of the second round. We suppose that such phenomenon can be explained by the fact that players were getting used to the game controls and rules in the first round, as well as learning how to play, which in turn had to some degree negative influence on their performance.

4.4 User Feedback

Since we decided to omit asking participants to provide subjective ratings of their user experiences, we observed their behavior and considered any feedback or comments they provided. Most of the players admitted that scoring a short-range interaction was considerably harder than performing a long-range one. In fact, most players considered that doing a punch was hardest due to the requirement of short distance between the interacting players. As we noticed, the players tended to perform rock-paper-scissors and waves first, because it was easier, and only then proceed with the short-range actions.

The majority of the players ($\sim 80\%$) noticed a strange correlation between the amount of actions they tried to perform and the resulting score. Most of them also attempted to inform us about possible bug in the implementation of the game model, since, as they claimed, they did not get the score for all the actions performed. The participants also mentioned that it was mostly happening when they tried to score a punch or, sometimes, a handshake. At the same time, we observed that while some players tried hard to get the required score, even though they noticed that not all of their action attempts were acknowledged, others were simply discouraged from performing short-range actions during the last rounds, getting an impression that they would not be able to fulfill that by any means.

Player behavior observations led us to the conclusion that increased latency had a negative influence not only on the objective score results, but also, on the players' subjective perception of the gameplay quality. Hereby, the players seemed to be somewhat frustrated while playing the last 2-3 rounds, since the application's response to their actions did not seem to meet their expectations anymore.

4.5 Summary

In this chapter, we presented our hypothesis and described the experiments that were performed to verify it. We hypothesized that short range-range player interactions were more latency sensitive than long-term interactions. In cases when player success was dependent on short-range interactions (like in the prototype we implemented), we expected that increased latency would cause player performance degradation and negatively influence the gameplay experience. Generally, results we derived from the experiments' outcome confirmed our hypothesis. We observed that the players' score, game duration, and winning probability were negatively affected by increased latency. The maximum tolerated latency (the highest delay at which the performance was generally not affected) for each action that we implemented in our test game model is listed in Table 4.4.

Table 4.4: Established tolerable latency thresholds

Category	**Action**	**Latency threshold**
Short-range	Punch	**100 ms**
	Handshake	**300 ms**
Long-range	Rock-paper-scissors	**$\sim$400 ms**
	Wave	**(at least) 1000 ms**

Chapter 5

Conclusion

5.1 Summary

In this work, we have investigated the effect of latency on short- and long-range player interactions. To address the problem, we have first studied the related work that focused on similar issues and derived some elements of the desired interaction model from the relevant literature. We figured out that in the related work, the principle of interaction ranges was not considered; at the same time, none of the works focused on determining the influence of latency on games that used human characters and human interactions characterized by various distance ranges. Therefore, we have implemented a prototype that included the missing parts. We have described the process of developing our test game model stepwise. While we proceeded with the implementation, different options for various parts of it were considered.

We described our solution for the 3D model of a human character, discussing different solution variants we considered, and evaluated two frameworks against their suitability for being used in our prototype - BGE and OGRE. Though BGE featured several useful components, it had somewhat inconvenient development interface, where there was no possibility to have full access to the source code; besides, the threading module could not be used efficiently, which was a known problem in BGE. OGRE appeared to be rather flexible and well documented, and was thereby chosen as the final solution.

While attempting to achieve the desired functionality in our implementation, we considered integrating a physics simulation library in our project. Bullet was one of the options that we evaluated, but for combining it with OGRE, we needed a wrapper. While we considered Bullet itself to be a rather useful and well documented framework, most of the wrappers did not provide clear documentation, which introduced a number of complications. A simple broad phase collision detection with the use of Bullet and OgreBullet was still a solution we looked into, but since it created much unnecessary overhead, we implemented a simple collision detection algorithm that only used OGRE instead.

We defined short- and long-range interactions in regard to their representation in our test model. To implement player interactions and the functionality they required, we considered different solutions for client side prediction and figured out that interpolation only was a better choice for our game model. We tried both spline and linear interpolation, while linear interpolation appeared to be more effective, in regard to our prototype. In addition, we implemented a simple method to correct score inconsistencies caused by latency.

Finally, we performed a series of experiments to establish the effect of network delay on each interaction we have implemented, as well as player performance and gameplay in general. We concluded that short-range interactions were considerably more latency sensitive than long-range interactions. Since short-range interactions were one of the factors that influenced player success, we noticed that player performance, winning probability, and subjective perception of the gameplay quality were negatively influenced by increased network delay.

5.2 Main Contributions

In this work, we have considered the literature that addressed relevant problems and implemented a prototype that included the issues that were not considered in the related work. Finally, we conducted user studies and figured out that distance range was an important factor in determining latency sensitivity of a particular player interaction. We established latency thresholds for each interaction that we implemented, based on obtained results. Generally, we concluded that short-range interactions can tolerate significantly lower latencies than long-range interactions.

5.3 Future work

This work requires further analysis in certain areas. Firstly, since we originally had an idea of creating custom animations for human characters during runtime, this issue may require some additional research. The functionality of steering different parts of the character's body can be easily implemented in our prototype, but an appropriate solution for the input system should be considered.

Secondly, there are certain optimizations that can be done to the existing interactions in our prototype. We have received some feedback from the participants regarding the speed of avatars' movements and rotations. Some users considered that the avatars could move and rotate a little bit faster. Therefore, further research can be done on whether the speed of movements and rotations would have an influence on the overall results.

Thirdly, we believe that performing experiments with a game model that the participants are well familiar with might influence the outcome. Thus, further investigation in this area is needed.

Appendix A

CD-ROM

The attached CD-ROM contains:

- source code for the final version of the test game model implementation (client and server)
- source code for the implementation of alternative solutions discussed in this work
- Blender files, containing the 3D-model and animations used in our project
- log files generated during experiments and used to derive the results
- installation instructions

A *README.txt* file in each folder (if applicable) provides the description of the contents.

Alternatively, the contents of the CD-ROM can be obtained from the following Git repository: git://git.uio.no/private-olgabo.git (Available until January 2013)

Bibliography

Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, November 2006. ISSN 0001-0782. doi: 10.1145/1167838.1167860. URL `http://doi.acm.org/10.1145/1167838.1167860`.

Pat Harrigan and Noah Wardrip-Fruin. *Second Person: Role-Playing and Story in Games and Playable Media*. MIT Press, 2010.

Andrew Rollings and Ernest Adams. *Andrew Rollings and Ernest Adams on Game Design*. New Riders Publishing, 2003.

Laurent Mathy, Christopher Edwards, and David Hutchison. Principles of qos in group communications. *Telecommunication Systems*, pages 59–84, 1999.

Tristan Henderson and Saleem Bhatti. Networked games: a qos-sensitive application for qos-insensitive users? In *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care?*, RIPQoS '03, pages 141–147, New York, NY, USA, 2003. ACM. ISBN 1-58113-748-6. doi: 10.1145/944592.944601. URL `http://doi.acm.org/10.1145/944592.944601`.

Douglas Comer. *Internetworking with TCP/IP*. Upper Saddle River, N.J.: Prentice Hall, 2000.

Grenville J. Armitage. An experimental estimation of latency sensitivity in multiplayer quake 3. In *ICON2003. The 11th IEEE International Conference on Networks*, pages 137–141, September 2003.

Tristan Henderson. Latency and user behaviour on a multiplayer game server. In *Proceedings of the Third International COST264 Workshop on Networked Group Communication*, NGC '01, pages 1–13, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42824-0. URL `http://dl.acm.org/citation.cfm?id=648089.747495`.

Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of loss and latency on user performance in unreal tournament 2003&#174;. In *Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*, NetGames '04, pages 144–151, New York, NY, USA, 2004. ACM. ISBN 1-58113-942-X. doi: 10.1145/1016540.1016556. URL `http://doi.acm.org/10.1145/1016540.1016556`.

Matthias Dick, Oliver Wellnitz, and Lars Wolf. Analysis of factors affecting players' performance and perception in multiplayer games, 2005. URL `http://doi.acm.org/10.1145/1103599.1103624`.

A. F. Wattimena, R. E. Kooij, J. M. van Vugt, and O. K. Ahmed. Predicting the perceived quality of a first person shooter: the quake iv g-model. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-589-4. doi: 10.1145/1230040.1230052. URL `http://doi.acm.org/10.1145/1230040.1230052`.

Peter Quax, Patrick Monsieurs, Wim Lamotte, Danny De Vleeschauwer, and Natalie Degrande. 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*, NetGames '04, pages 152–156, New York, NY, USA, 2004. ACM. ISBN 1-58113-942-X. doi: 10.1145/1016540.1016557. URL `http://doi.acm.org/10.1145/1016540.1016557`.

Sebastian Zander and Grenville Armitage. Empirically measuring the qos sensitivity of interactive online game players. In *Australian Telecommunications Networks and Applications Conference*, pages 511–518, December 2004. ISBN 0-646-44190-6.

Tobias Fritsch, Hartmut Ritter, and Jochen Schiller. The effect of latency and network limitations on mmorpgs: a field study of everquest2. In *Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*, NetGames '05, pages 1–9, New York, NY, USA, 2005. ACM. ISBN 1-59593-156-2. doi: 10.1145/1103599.1103623. URL `http://doi.acm.org/10.1145/1103599.1103623`.

Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. Effect of network quality on player departure behavior in online games. *IEEE Trans. Parallel Distrib. Syst.*, 20(5):593–606, May 2009. ISSN 1045-9219. doi: 10.1109/TPDS.2008.148. URL `http://dx.doi.org/10.1109/TPDS.2008.148`.

Lothar Pantel and Lars C. Wolf. 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, pages 23–29, New York, NY, USA, 2002a. ACM. ISBN 1-58113-512-2. doi: 10.1145/507670.507674. URL `http://doi.acm.org/10.1145/507670.507674`.

Mark Claypool. The effect of latency on user performance in real-time strategy games. *Comput. Netw.*, 49(1):52–70, September 2005. ISSN 1389-1286. doi: 10.1016/j.comnet.2005.04.008. URL `http://dx.doi.org/10.1016/j.comnet.2005.04.008`.

Gang Li, Zheng Wu, Xiaoli Meng, and Jiankang Wu. Modeling of human body for animation by micro-sensor motion capture. In *Proceedings of the 2009 Second International Symposium on Knowledge Acquisition and Modeling - Volume 03*, KAM '09, pages 98–101, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3888-4. doi: 10.1109/KAM.2009.174. URL `http://dx.doi.org/10.1109/KAM.2009.174`.

Stefan Zerbst. *3D Game Engine Programming (Game Development Series)*. Premier Press, 2004. ISBN 1592003516.

Len. Fisher. *Rock, paper, scissors: game theory in everyday life*. Basic Books, 2008. ISBN 9780465009381.

D.H. Eberly. *Game Physics*. Interactive 3d Technology Series. Elsevier Science, 2003. ISBN 9781558607408. URL `http://books.google.ca/books?id=a9SzFHPJ0mwC`.

C. Ericson. *Real-Time Collision Detection*. Number v. 1 in Morgan Kaufmann Series in Interactive 3D Technology. Elsevier, 2005. ISBN 9781558607323. URL `http://books.google.no/books?id=WGpL6Sk9qNAC`.

Yahn W. Bernier. Latency compensating methods in client/server in-game protocol design and optimization, 2001. URL `https://developer.valvesoftware.com/wiki/Latency_Compensating_Methods_in_Client/Server_In-game_Protocol_Design_and_Optimization`.

G. Junker. *Pro OGRE 3D Programming*. The Expert's Voice in Open Source Series. Apress, 2006. ISBN 9781590597101. URL `http://books.google.no/books?id=GifUrbWat14C`.

E. Catmull, , and R. Rom. A class of local interpolating splines. *Computer Aided Geometric Design*, pages 317–326, 1974.

M. Mauve, J. Vogel, V. Hilt, and W. Effelsberg. Local-lag and timewarp: providing consistency for replicated continuous applications. *Multimedia, IEEE Transactions on*, 6(1):47 – 57, feb. 2004. ISSN 1520-9210. doi: 10.1109/TMM.2003.819751.

Lothar Pantel and Lars C. Wolf. On the suitability of dead reckoning schemes for games. In *Proceedings of the 1st workshop on Network and system support for games*, NetGames '02, pages 79–84, New York, NY, USA, 2002b. ACM. ISBN 1-58113-493-2. doi: 10.1145/566500.566512. URL `http://doi.acm.org/10.1145/566500.566512`.

Amir Yahyavi, Kévin Huguenin, and Bettina Kemme. Antreckoning: dead reckoning using interest modeling by pheromones. In *Proceedings of the 10th Annual Workshop on Network and Systems Support for Games*, NetGames '11, pages 1:1–1:6, Piscataway, NJ, USA, 2011. IEEE Press. ISBN 978-1-4577-1934-9. URL `http://dl.acm.org/citation.cfm?id=2157848.2157850`.