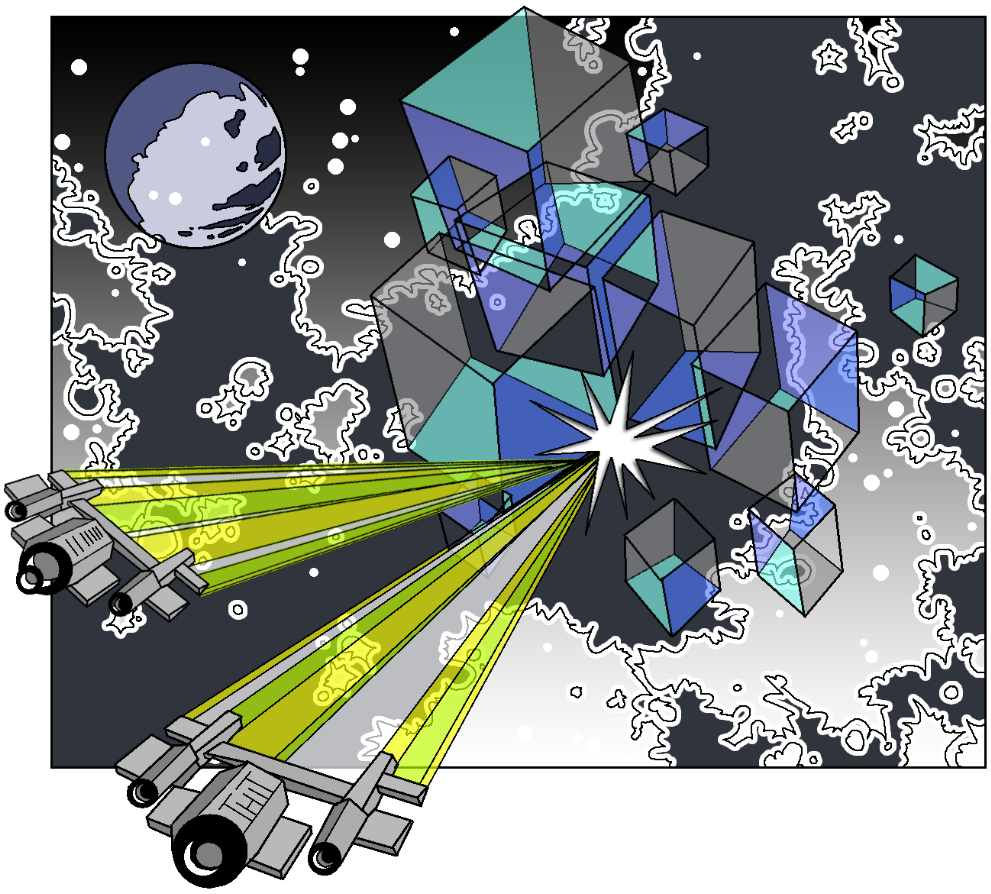
To Infinity and Beyond

Scaling Massively Multiplayer Games



Master Thesis by
Ron Cohen
Anders Ejlersen
Rasmus Kristensen

Aalborg University, May 31st, 2010

AALBORG UNIVERSITET

Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
Phone 96 35 80 80, Fax 98 15 97 57
http://www.cs.aau.dk

Title:
To Infinity and Beyond: Scaling Massively Multiplayer Games

Theme:
Distributed Systems

Project period:
DAT6, spring 2010

Project group:
d619a

Authors:
Ron Cohen
Anders Ejlersen
Rasmus Kristensen

Supervisor:
Brian Nielsen

Print count: 5

Nr. of pages: 96

Appendix count and -type:
6 pages and 2 DVD's

Completed & signed:
May 31st 2010 at Aalborg University

Abstract:

This report pertains to load balancing techniques for use in MMOGs with focus on geographical partitioning techniques. An analysis of existing two-dimensional techniques is performed, namely: Static-, Matrix- and JoHNUM- partitioning. We have extended these techniques to be applicable in three-dimensional virtual environments. Furthermore, we have created a test game called Rock Pounder, which has been used to evaluate the extended techniques. The evaluation is performed with two different scenarios, which represent common MMO game play scenarios. From the evaluation we found that the static partitioning technique was well suited for static work loads. However, for dynamic work loads we found the dynamic JoHNUM techniques to be the most suitable, mainly due to the more involved pre-partition analysis of the virtual world. It was found that JoHNUM in particular yields a fairly good scalability regarding number of supported players relative to the number of servers operating.

Contents

Preface

This report has been written during the Dat6-project period, by computer science group d619a at Aalborg University. The area of the report is in "Distributed and Embedded Systems" and therefore mainly consists of knowledge from that area. This report is addressed to other students, supervisors, and anyone else who might be interested in the subject. To read and understand the report correctly, it is necessary to have knowledge about the most basic computer and game related terms.

The entire report is written in English and no translation will be accessible, it is therefore required to understand English. Abbreviations and acronyms will at first appearance be written in parenthesis, to avoid breaking the reading stream. Furthermore a list of game terminology is found in Appendix A and an e-mail correspondence with Umar Farooq in Appendix B. Specification of gender in the report is not to be understood as suppression or any other form of political/religious position. The gender is only specified to simplify the process of writing for the authors.

References to sources is marked by [#], where # refers to the related literature in the bibliography at the end of the report.

The appendix to the report is found in the last chapter of the report, and on DVDs, located on the very last page of the report.

The report is written in LaTeX and is accessible as a PDF-document.
Special thanks to Thit Hansgaard for creation of the front page image.

Signatures:

Ron Cohen	Anders Ejlersen	Rasmus Kristensen

Chapter **1**

Introduction

Multiplayer games have seen an amazing development within the last couple of decades. From two players sharing a computer, a keyboard, and a monitor to several players on distant computers. In the recent years, a new type of multiplayer game has seen the light of day: Mass-Multiplayer Online Game (MMOG). These types of games have thousands of players participating in the same virtual world, simultaneously. Having this extraordinary number of players interacting in the same virtual world makes it possible to invent new interesting types of game play, pushing game play beyond the limits of traditional multiplayer games. EVE Online[1] is an example of a game belonging to the MMOG genre. In EVE Online all players participate in the same virtual world. In this virtual world it has been possible for thousands of players to develop a virtual macro economy, and it is now a significant feature of the game. With too few players, the virtual macro economy becomes fragile and susceptible to manipulation. EVE Online's virtual macro economy is an example of game play features, which can only be achieved in virtual worlds which have a lot of players participating. Features like these can be very important for the commercial success of a game, however, creating a system which is able to sustain an enormous number of players at any given time poses a significant engineering challenge.

A few current MMOGs can support as many as 180,000 players in the same virtual world[2]. At this level, ensuring a good game experience becomes a major challenge due to the technical difficulties which exist in highly distributed, dynamic systems that modern MMOGs are. Specifically, as the number of players in the same virtual world increases, so does the overall complexity of the virtual world and the total rate of interaction in the virtual world. With hundreds or thousands of players, the rate of interaction across the virtual world becomes enormous and the resulting workload constitutes a very substantial problem, which will only become more severe as the number of players increase.

To deal with the issue of handling a vast number of players and the interactions due to the number of players, a distributed system is used. However, it is non-trivial to distribute the workload in a manner that enables the solution to take advantage of additional computing resources to serve the increase in number of players.

Today, MMOGs are predominately client/server based and most computation related to the virtual world is performed by the servers. Each player connects to the server using a client. We assume that each client controls one avatar at a time in the virtual world. For each interaction happening in the virtual world, there is a related consequence of the interaction which must be computed to update the virtual world. Afterwards the information about the interaction must be sent to applicable clients, as illustrated in Figure 1.1. These are typically clients controlling avatars which are adjacent in the virtual world to the location where the interaction took place.

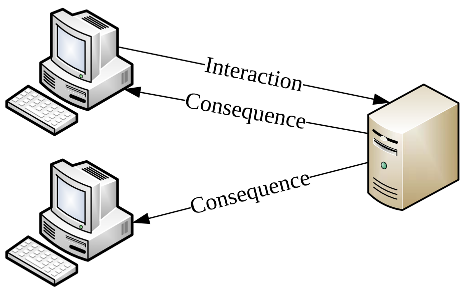


Figure 1.1: Each interaction is sent to the server. The server calculates the consequences and disseminates the updated virtual world back to the clients.

The two games: World of Warcraft and EVE Online are both in the category of MMOGs. Both games use *targeted interaction*, in which a player must specify which entity on the screen she wants to interact with before interaction can begin. However the interaction in First-Person Shooter (FPS) games is different, because the player uses a mouse or a joystick to point their virtual weapon, thus taking aim at the objects they want to strike. This is called *free interaction.* Targeted and free interaction type games tolerate different amounts of latency. The latency requirements in targeted interaction type games are not as strict as the free interaction type games, where tolerances for latency are much lower. This is likely the reason that many MMOGs use the targeted interaction type game play[3].

In games from the FPS genre, a player can quickly interact with several objects within a very short time period. This is because interaction is not restricted as in many other MMOGs. FPS games use this kind of interaction to allow players to experience a quick and fast paced game play, where speed and quick reaction is key to achieving the objectives in the game. This is the reason why FPS games have a typical limit of 50-100ms of latency[3], where targeted interaction could have upwards of 200ms of latency[4].

1.1 Related Work

Research has gone into the distribution of workload in MMOGs and virtual environments. These have both been used in popular games, like World of Warcraft[5] and EVE Online, where others are either still in the research state or not mentioned in any products, because of their commercial value.

In most smaller multiplayer games the amount of players is limited to tens and the server application is designed to run on a single machine. Every client is notified about everything that occurs in the virtual world, this is known as *global consistency.* It is important that clients receive information about the state of the virtual world, since no player should have an unfair advantage against other players, because she has a view of the virtual world that is more up-to-date, than the others. However MMOGs use *localised consistency* instead of global consistency, in order to avoid that all players must know about all other objects. The reason why MMOGs have opted for localised consistency is typically because the virtual world in a MMOG is of much greater scale, than other types of games. For a MMOG it is infeasible for all clients to have up-to-date information about the whole virtual world, and all the objects it is compounded of. Therefore most known load distribution techniques exploit localised game play and use this game play property to provide localised consistency.

Localised game play restricts the interactions between objects, to objects which are located near each other in the virtual world. That is, objects can only interact with objects in their immediate vicinity. This property makes it possible to provide localised consistency while still giving the illusion of global consistency. That is, since players in games have a limited *field of view*, it is actually only necessary to inform the player about objects within its field of view. A method to effectuate this is if the virtual world is partitioned into geographical regions, e.g. Europe is partitioned into countries and countries can be partitioned into counties, each of these can in a virtual world be a geographic region, where each geographic region can be handled by a server.

In our DAT5-project[6] we covered some of the different solutions available. We described both single-server and multi-server solutions, using either static or dynamic virtual world partitioning. The multi-server static partitioning technique is believed to be used in most popular MMOGs, such as EVE Online to cope with the large number of players. The dynamic partitioning techniques attempts to improve upon this by dynamically partitioning and merging the virtual world according to predefined partitioning schemes and load evaluators. The two main techniques mentioned in our DAT5-project were Matrix[7] and Joint Hierarchical Nodes based User Management (JoHNUM)[8, 9]. Both use dynamic virtual world partitioning to load balance the workload. The original Matrix and JoHNUM articles explain the design and evaluates the techniques using a simulation. However, to the best of our knowledge, no working prototype of these techniques have previously been developed which can be used to compare the performance of the techniques. Matrix has been implemented in test games, however, in order to compare performance versus the JoHNUM technique a general prototype, encompassing both the Matrix and the JoHNUM technique is required.

1.2 Project Goal

When virtual worlds increases in size, and the number of virtual objects grows to the size of MMOGs, a distributed solution becomes a necessity in order to provide an acceptable game experience.

Load distribution across a distributed solution in MMOGs is often done by partitioning the virtual world and letting a server handle each partition. However, partitioning a virtual world onto several servers raises the issue of how to give clients a consistent view of the virtual world. MMOGs solves this by exploiting localised game play and offers only localised consistency to players, as Section 2.2 explains, this suffices to give an enjoyable game experience.

1.2.1 Problem Definition

This reports concerns the problems of distributed game engines for use with MMOGs, which have special latency requirements similar to those of FPS games. It therefore concerns the problems of distributing the workload, while maintaining a consistent view of the virtual world, and how to optimise load distribution such that server utilisation is as optimal as possible. Additionally, the partitioning techniques mentioned in Section 1.1, only concerns load distribution of a virtual world by partitioning it in a two-dimensional fashion, however, certain virtual environments do not have the property of only partitioning it in two dimensions, but require three-dimensional partitioning. Further, Matrix and JoHNUM have, to the best of our knowledge, never been implemented and compared in the same prototype. The following problems will therefore be examined in this report:

- How to generalise existing techniques from a two-dimensional to three-dimensional virtual world?
- How do the three-dimensional techniques compare against each other?
- How close to linear speedup can be achieved by using these techniques?

The test game: Rock Pounder, that was partly developed during the DAT5-project period[6] will be utilised during design, implementation, and test of various load distribution techniques. The test game will as in the DAT5-project have a playable version of the game, but also a client for simulating many clients simultaneously. The test game evolved during this project period from a single server to support multiple servers and multiple partitioning techniques.

1.3 Report Structure

This report contains an analysis in Chapter 2, that examines the different factors that are important for a good game experience, which network architectures that can be used in MMOGs and which are used in commercial games. Then different load distribution approaches by geographic partitioning of a virtual world will be described.

This is followed by Chapter 3 pertaining to the design of distributed solutions for MMOGs, specifically Rock Pounder. This chapter handles the load distribution solutions from the analysis, how they can be designed for use with Rock Pounder and specifically to handle the problems mentioned in Section 1.2. The first solution is a static multi-server solution known from popular games like EVE Online. The second is Matrix, which is a solution that challenges the static multi-server solution by applying dynamic partitioning and merging techniques in the virtual world. This solution have further been extended by JoHNUM, but both JoHNUM and Matrix focuses on partitioning and merging of a two-dimensional environment. They are therefore extended such that they can be used within a three-dimensional environment.

A general game engine for use in games is designed and described in Chapter 4. This game engine is used in Rock Pounder and contains a description of the different parts of the engine. This includes the various components that a game engine has, and a detailed description of the game loop, which is the key component in a game engine. Then Rock Pounder is described. Game play and design of the game is discussed, followed by a discussion of why Rock Pounder is applicable as a test game for the different load distribution solutions detailed in Chapter 4.

The different solutions are tested and evaluated in Chapter 5. The static multi-server solution is evaluated against a single server solution. It is then used as the benchmark for the Matrix and JoHNUM techniques. They are compared on two test scenarios, which consists of two different battle scenarios.

Finally, a conclusion is given. The different key aspects discovered in the test chapter will be summarised and concluded upon, and lastly different areas of future work are discussed.

Chapter **2**

Analysis of Load Distribution in MMOGs

This chapter gives an overview of the typical components of a game engine. Then an analysis of the factors which impact the game experience is given. Followed by an analysis of different network architectures for use in MMOGs. Then an analysis of three different techniques for load distribution in MMOGs, namely *static virtual world partitioning*, *Matrix*, and *JoHNUM*. This is followed by an analysis of the characteristics of a theoretical optimal load distribution solution.

2.1 Game Engine

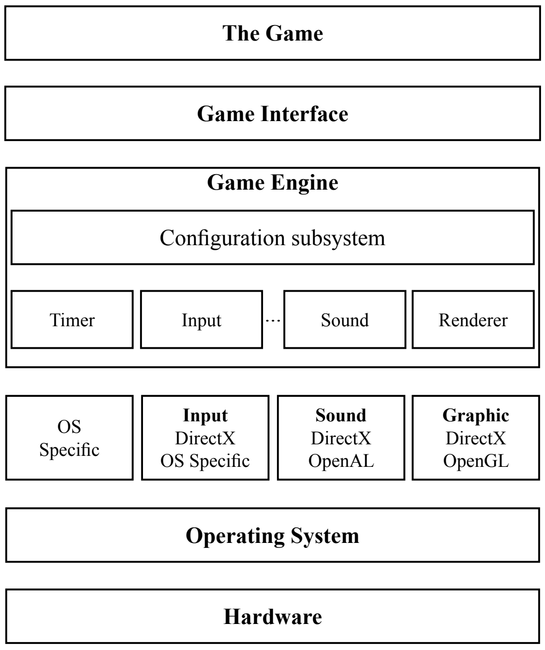
The game engine is the foundation of a game. It is a framework which a game can be build upon. The game engine can be created in a more or less generalised manner, depending on the requirements of the game. An example of a generalised game engine is Unity 3D[10], which was created for the purpose of providing a foundation for independent game development. However, large commercial games like EVE Online are typical build on their own game engine[1], as they have very specific requirements to what the game engine must be capable of.

2.1.1 Components

The primary task of the game engine is to provide the necessary components to create a game[11]. A game engine structure is illustrated in Figure 2.1 with respect to creating a whole game.

The game engine is a layer between the game and the lower level API's, like playing audio and drawing to the screen. This basic level functionality is represented in the engine layer in Figure 2.1. The engine hides the lower level API's, e.g. OpenGL from the game code. In addition to these basic components a game engine consist of a number of game specific components. These components vary with the purpose of the engine, but a list of common components are presented here:

Virtual Object Concept: The objects in the virtual world (also called virtual objects) should be able to be maintained by the game engine. A virtual object can be manipulated by the game logic to conform with the game rules, but the engine should still be



[Illustration from DAT5 report[6].]

Figure 2.1: The layered structure of a game, where the bottom layers are system specific, the middle layer are the game engine and the top layers are the game specific code.

able to, e.g. apply physics to the virtual object. These virtual objects can be players, projectiles, and other virtual objects in the game.

Physics: When a virtual object, e.g. a barrel, is inserted into the game it is expected to follow the laws of physics to some degree. To handle this, a physics system is often a part of the game engine. One fundamental function of physic systems is collision detection. Collision detection is used to determine when and where virtual objects collide, such that the physics system can take the appropriate actions to have the object move as expected.

Artificial Intelligence: In many games there are a Non-Playable Character (NPC) which human players can play with or against. Basic features, like pathfinding and sensory systems, is usually a part of the engine. However, higher level logic is handled in the game logic, where it can be desirable to have some specific events happen for the NPC. This can be to take the role of telling a story in the game, and then at specific points in the game the NPC should perform pre-scripted events.

Effect Systems: The game engine is, as previously mentioned, responsible for exposing graphics and sounds such that it can be used in a game. On top of the basic capabilities, complex effects like explosions with particle systems and special sound effects are added to be able to ease the use of these complex effects.

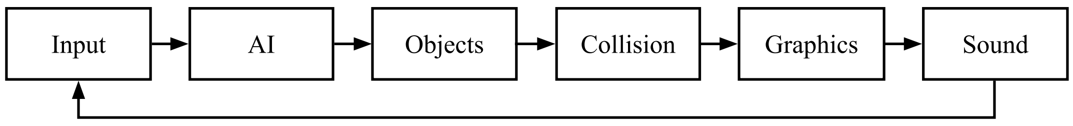
The code which describes the game logic is written in a high level of abstraction. The high level code utilises the components which is exposed in the game engine. It is desirable to specify game logic in a high level fashion, because game logic is frequently changed during a game's development. As a game can go through many development iterations to ensure that the game is fun to play.

Naturally, some components of the game engine use more system resources than others. Often, physics and collision detection is a big part of games, and a lot of resources goes towards these components. Other components which often use a lot of resources are the AI and game logic components, which needs to take complex decisions.

An important concept in game development is the *game state*. The game state is any data that relates to the state or configuration of the virtual world. For the most part, the game state consists of the virtual objects of the virtual world and properties of those, such as velocity, size, position, and so on. One could also imagine that components like the AI components may have some state. Different components manipulate the game state in different ways. The physics component updates the virtual objects locations, according to the velocity of each object, thereby altering the game state.

2.1.2 Component Structure

The components mentioned in the previous section are executed in an order to make the virtual objects "come alive". There are a number of elements which is needed before the virtual objects can be updated. An example of a game loop with AI, collision detection, and so on, is illustrated in Figure 2.2.



[Illustration from DAT5 report[6].]

Figure 2.2: The typical components in the game loop of a game, which is executed in sequence.

All of these components run in a loop called the *game loop*. Each iteration of the game loop is called a *frame*, as a game loop iteration results in a new frame being drawn on the screen. The game loop is executed frequently to give a real-time feeling of the game. The frequency is game dependent, but if there are continuously moving elements, a frame rate around 30 frames per second is necessary to make it look fluent, as it is with motion pictures. *Frame rate* is measured in *frames per second*.

The first component in Figure 2.2 is the input from a player, this could be a key that is pressed or a mouse movement. These actions are used later to determine what the virtual objects in the virtual world should do, as the player interacts with them. Then the NPC's actions are calculated, this could be compared to the players input, as the inputs affects the virtual objects in the virtual world. When all the inputs are collected the new game state is calculated. The changes to the game state could be virtual objects that moves, are destroyed, or new virtual objects that has been created. Afterwards, the collision detection is invoked to find collisions between virtual objects and the results to these are applied. This could, e.g. be a ball that have just been hit by a bat. Lastly, all the virtual objects are drawn onto the screen and the appertaining sounds are played.

The game loop structure can however change if it is, e.g. a multiplayer game. Then there can be a server which handles the virtual objects, AI, and collision detection and sends the game state to the clients, which then draws the game state. This can be done in an asynchronous manner, as it is not desired for the server to wait for all the clients in a multiplayer game to draw the state before continuing, as it could ruin the real-time experience of the game.

2.2 Game Look and Feel

In order to be able to evaluate load distribution techniques, it is important to understand which factors impact the game experience for the players.

Game designers strive to give players an *immersive* experience. What characterises of the immersive experience is that players become absorbed with the game, they may loose the sense of time while playing, or perhaps even think about the game while away from the game. Good games continue to give players an immersive experience every time they are played.

Achieving the goal of the immersive experience is a science in itself and succeeding depends on many factors, such as the ability of the game to maintain the players interest. This can be done with posing interesting challenges, showing breathtakingly beautiful graphics, telling an exciting story, and so on. Of course, different player demographics also play a role in this equation. For example, senior players may find some challenges uninteresting, while young children can find the same challenges interesting. Female players may like different game types than male players, and so on.

In multiplayer games two issues are especially prone to ruining the immersive experience. One of these is making sure players see the same representation of the virtual world at any one time. After all, there should be only one true configuration of the virtual world at any one time, e.g. virtual objects can only be in one location at any time. Failing to do so may be very confusing for players as they will be unable to understand why other players are acting the way they are, because they see the virtual world differently. There are two sides to this: It is important that players *share the view* of the virtual world. However, assuming that a server is the authority on the state of the virtual world, it is important that players receive correct information from the server about the state of the virtual world. This is further complicated when many servers cooperate to represent the virtual world, as it is very important that clients do not receive contradictory information about the state of the virtual world. Providing *localised consistency* fulfils these requirements, Section 2.2.1 will elaborate on this.

The second part is that the changes in the virtual world are shown on player screens within some time-limit. In situations with high latency, changes made to the virtual world may take a long time to become apparent for players. A consequence of this is that players can experience a very delayed response to their actions. This has the potential of making the experience a lot less enjoyable as the player will feel the game is not a smooth experience. The amount of latency that is tolerable is very much game dependent. Some turn- or strategy-based games (e.g. a chess game) can have latencies up to several seconds without harming the immersive experience. On the other hand, FPS games are most often quite sensitive to latency and typically require a low latency to give an enjoyable experience, as mentioned previously. One of the causes of latency is a server becoming overloaded. Servers are typically built such that they run a specific set of instructions a number of times per second, called the *game loop*. If a server becomes overloaded, it will not be able to execute these instructions at the rate it was designed for. In most cases it will degrade gracefully and try to maintain an acceptable level of service until the workload is remedied.

Latency can also be a source of unfairness in real-time multiplayer games. Some players may have much higher latency than others, because of their network connection and thus, messages regarding the state of the virtual world reach them later. In FPS games this can result in a significant disadvantage for players with high latency, and thus make the challenges posed by the game too difficult to achieve the immersive experience.

2.2.1 Localised Consistency

The immersive experience is important for the clients and therefore needs to receive the correct information in a timely manner, as mentioned in Section 2.2. This section explains what information is required by which clients in order to provide localised consistency. We adopt a definition of localised consistency akin to that of Matrix[7].

Localised consistency is provided on the premise that the virtual world is subject to natural geographic features. It must be possible to calculate the distance between objects and objects can only be observed by avatars (virtual characters) if the distance between them is within some limit. There are two ways to define this limit:

Radius of visibility: A radius surrounding an object within which they can be observed. This model allows for small objects to have different radius of visibility than larger objects.

Radius of sight: Avatars have a fixed radius of sight. Any objects within this radius is observable for the avatar.

In essence, the difference between the two is the range in which an object can *be* observed, and the range in which an avatar *can* observe other objects. For simplicity, we use *radius of sight*. The radius of sight around an avatar is the radius for which local consistency must be provided for the client controlling that specific avatar in order to give a coherent game experience.

Region partitioning techniques partitions the virtual world, Z, into N different non-overlapping partitions $\{P_1, P_2, P_3, ..P_N\}$, and assigns each partition P_i to a server S_i. With the dynamic region partitioning techniques, N can be changed dynamically to accommodate for load fluctuations. The partition managed by server S_i can also be changed dynamically.

Changing game state at any point σ_i of the virtual world Z, handled by server S_i, which is within the range of sight of any avatar on server S_j, requires the new state to be propagated to the avatar on S_j as well as any avatars within range of sight on S_i.

Given a radius of sight R, every point σ in the virtual world, Z, has a set of servers associated with it, called the *consistency set* of σ or $C(\sigma)$. This set consists of all the servers whose partitions overlap with the circle with radius R centered in σ. Avatars residing on servers in $C(\sigma)$ may need to know about events in σ, thus servers in $C(\sigma)$ should be updated when the state in σ changes. Formally, the consistency set is defined as:

$$C(\sigma \in P_i) = \{S_j \mid j \neq i \wedge \exists \sigma' \in P_j \ s.t. \ d(\sigma', \sigma) \leq R\}$$

If R is set to infinity, every server must be updated when a state changes. This makes it impossible to provide efficient localised consistency. However, if R is relatively small compared to the size of partitions P_i, then most points should have $C(\sigma_i) = \emptyset$, thus only requiring S_i to know about events in σ_i.

An *overlap region* is made up of points sharing the same non-empty consistency set. Intuitively, an overlap region describes an area of the virtual world in which occurring events must be propagated to servers handling adjacent regions. The consistency set is thus useful for determining which servers should receive updates.

If updates are propagated using UDP packets, they are as such, not reliably transmitted. Further, there is no ordering guarantees which can lead to packets arriving out of order. This is the case for both server to server updates as well as server to client updates. This means information transmitted to clients may be outdated momentarily if a packet is lost or a packet arrives out of order, however, as each update contains the entire state of an object it will not be a problem after the next update is received. Similarly, commands from clients to servers may be lost in transit. Again, as updates are transmitted very frequently and each update contains the complete state information, some lost packets will affect the game experience on a negligible level. Obviously, if a significant number of packets are lost, it will affect the game experience negatively.

This may seem like a very weak approach to consistency, however, in fast-paced games such as most FPS games, strict consistency is overlooked in favour of quick dissemination of updates. Due to the fast-paced nature of these games, players will rarely, if ever, notice the small errors which can occur due to inconsistent updates.

2.3 Network Architectures

This section describes some of the preferred network architectures used in MMOGs. This is the peer-to-peer and client/server approaches, which are compared. Then a generalised Mass-Multiplayer Online (MMO) architecture is described, which is constructed of the knowledge learned from peer-to-peer and client/server approaches learned during the analysis, but also regarding the approaches used in commercial MMOGs today.

2.3.1 Peer-to-Peer

In a peer-to-peer network, every participant is an equal peer in the network. All peers uses the same program and are both a client and a server at the same time. The peer-to-peer topology in which peers exchanges messages directly with each other, is illustrated in Figure 2.3 (a).

The peer-to-peer approach was the dominant approach in early multiplayer games. These games include *Doom* and *Duke Nukem*, which both are FPS games. These games were popular before the internet became ubiquitous and were played over modem or local area networks.

Peer-to-peer systems are used today extensively for file-sharing purposes. Popular peer-to-peer based file-sharing networks include *BitTorrent* and *Gnutella* networks.

One of the advantages of a peer-to-peer based approach, as opposed to a client/server based approach, is that computing resources are added automatically for each participating player. From the game producers point of view, it is also a very inexpensive method to service players, because the number of costly control servers can be minised in a peer-to-peer based approach.

A significant disadvantage of peer-to-peer based approaches with relation to games, is that cheating can be very hard to detect. This is a problem on two levels:

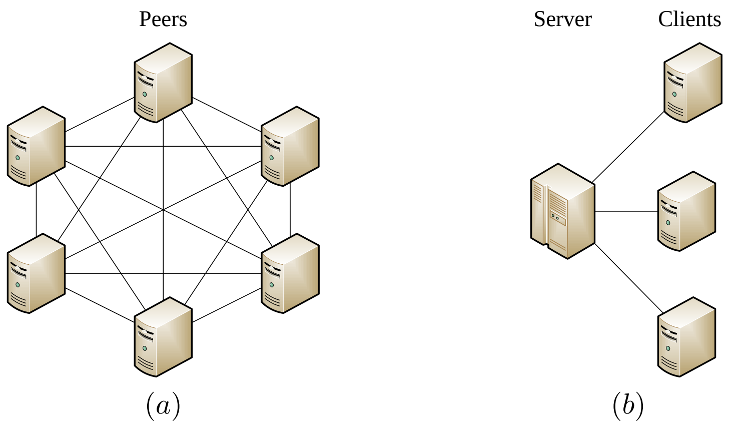
- In-game assets such as currency, virtual goods, and so on, are hard to control when everything is stored on the players computers. Players may manipulate local storage to their advantage.

- It can be hard to ensure a fair and enjoyable game environment, because peers send data directly to other peers. Players may alter the code that they run in order to send false information to their peers or withholding information to some peers, thus hindering fair and enjoyable play.

Additionally, it is desirable for game designers to maintain control over who plays their game. Controlling which players are connected to the game can also be difficult in peer-to-peer based approaches.

Besides the issue that exist with cheat and control, peer-to-peer based architectures also suffer from well-know problems that are common to peer-to-peer based internet applications today, including firewalls and NAT-traversal problems. Assuming players typically play from their home, they will often be connected to the internet through DSL or cable connections. In order to access the internet from more than one computer, a Network Address Translation (NAT)-device must be used. In peer-to-peer based systems it can be hard to set up a connection between two computers when the connection must be established through NAT-devices on each end of the connection. Further, personal firewalls may hinder game related network services operating on personal computers[12].

These problems severely complicates any peer-to-peer based solution.



[Illustration from DAT5 report[6].]

Figure 2.3: Topology for (a) Peer-to-peer network and (b) Client/server network.

2.3.2 Client/Server

In a client/server based approach computers have either the role of: Client or server. Servers runs applications which puts services at the clients disposal. The clients connect to servers, which then allows clients to use the services provided, as illustrated in Figure 2.3 (b). Clients initiate connections to servers, thus, servers *listen* for incoming connections. Conceptually, servers offer their resources to clients. Clients do not contribute resources to the network, they consume resources.

An example of a client/server based system that most people know is *the web*. Browsing the web involves directing client software (e.g. *the browser*) to connect to a web-server, sending a request (usually using the HTTP protocol[13]), and receiving a response.

The client/server approach has been used throughout gaming history. Games like Quake III Arena[14] used a client/server architecture where one machine was designated the server, while the clients would send their input to the server and receives the game state to render. The server uses the input from the clients to allow some actions to take place, e.g. movement or interaction with other players.

In a client/server based approach, the game engine is split across the server and clients. Most components still execute on servers, while clients accept keyboard inputs and render a representation of the virtual world to players. In order to correctly render the virtual world, many objects of the virtual world are replicated to clients. This require frequent updates to be sent to clients to keep their replicas current. The exact messages that are transferred between clients and servers are very much game dependent.

In the previous section it was mentioned, that a client/server based approach requires servers to be maintained by game producers. This can be an expensive and cumbersome task, so a peer-to-peer approach might be preferred. However, the client/server approach does not suffer from the firewall and NAT traversal issues mentioned in the previous section.

2.3.3 Hybrid

A hybrid between the client/server and the peer-to-peer approach can be created. Some of the most popular file-sharing networks have used a hybrid architecture. The file-sharing network *Napster* used central servers for indexing, while still performing file transfers directly between peers[15]. Peer-to-peer based solutions makes sense when it is possible to utilise the resources that peers can contribute to the network.

In real-time games, the large majority of data communication consists of object updates. In order to reduce the communication cost for servers, one could imagine that peers could help transmit these object updates to other peers. However, this involves trusting the peers to disseminate the correct information and in a timely manner.

We have found that the most significant CPU load on servers comes from performing collision detection between objects in the virtual world. Utilising the resources that peers can contribute with, could be used to perform collision detection, thereby easing the load on the game servers. However, the core of the problem remains the same. It would be necessary to trust the peers to do correct collision detection. Collision detection plays a significant part in determining if a player has fulfilled the game objectives in, e.g. FPS games. Tampering with collision detection could yield substantial unfair gains for some players.

Many games have used a hybrid approach in that one of the players' machines was assigned to be the server, and the other participants were clients. Games using this approach include Doom, Counter-Strike, and StarCraft amongst others[16]. These games were played with 2 to 32 players in the same virtual world at a time. Anyone was free to start a new virtual world, and thus, making ones own machine the server (also know as the *host*). The fact that people were free to start their own virtual world made the problem of cheating less significant compared to cheating in an MMOG context. If you suspected a host of having an unfair advantage, one could simple switch to another host. The concept of MMOGs is that many players participate in the same virtual world, thus, switching virtual world due to cheat is not an option.

Finally, assuming that a hybrid between the peer-to-peer and the client/server architectures is created, it would still require peers to communicate with each other directly and therefore deal with the aforementioned NAT and firewall problems in addition to the trust related issues already mentioned.

For the reasons mentioned, a hybrid was deemed unfit for MMOGs, and real-time MMOGs in particular.

2.3.4 Network Architectures & MMOGs

The predominant network architectures available are client/server, peer-to-peer, and hybrids of these, as described above. However, the client/server approach is preferred in commercial MMOGs due to the following requirements:

Cheat Prevention: All data is passed to official trusted servers, which ensures that input from clients conform to the rules of the virtual world.

Control: Due to the fact that many MMOG are subscription based, and that it is desirable to be able to evict players for different reasons, it is important for MMOG creators to maintain some degree of control over who plays the game.

These requirements can be hard to realise in peer-to-peer based architectures, as opposed to client/server architecture, as mentioned in the previous sections. Due to the fact that the responsibility is distributed onto all connected peers, control becomes difficult, since there is no centralised mechanism to maintain this control. Finally, cheat prevention is also difficult since there are no centralised mechanism to check for malicious data from other peers[17].

2.3.5 Generalised MMO Architecture

We presented a generalised MMO architecture in our DAT5-project[6], which is illustrated in Figure 2.4. Game producers typically reveal very little about the network architecture used in their games, however, quite a lot of information has been made available about EVE Online[18]. The generalised MMO architecture has been compiled from the analysis of EVE Online, a very popular MMOG. It consists of three layers: Client, application (compounded of several sub-layers), and persistence layer.

Client Layer: Consists of the clients currently participating in the game. Clients connect through the internet to the application layer. The responsibility of the clients is to act as the interface between the player and the application layer. Clients will take input from the player and forward it over the internet to the application layer - and receive updates from the application layer and present these to the player.

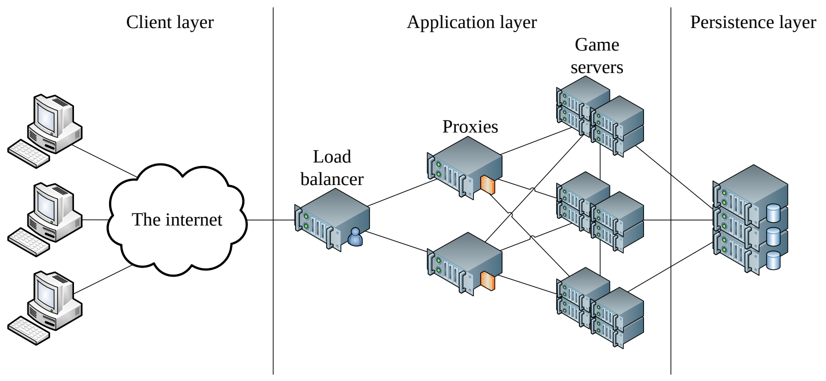
Application Layer: Responsible for receiving input from clients, processing the input, and sending updates to clients. This layer can be very complex and consist of several sub-layers, such as:

- **Game Servers:** Apply the actual game logic to the game state, in addition to physics, collision detection, and so on.
- **Load-balancers:** Responsible for balancing the work load across the machines in the application layer.
- **Proxies:** Responsible for disseminating the updates back to clients and forwarding updates from clients to the intended machine in the application layer. They also assist in creating a uniform view of the server architecture for clients and hiding complexity from the clients.

Persistence Layer: Used to allow application layer machines to save information for later use. Often, the persistence layer is presented to the application layer in a coherent and uniform way in order to simplify the application.

The generalised MMO architecture is quite akin to the well-known 3-tier, having presentation, logic, and data tiers. In the MMO architecture, the state is maintained and manipulated in the middle layer. With a 3-tier architecture, it is normal to have the logic layer manipulate data residing in the data layer[19]. The real-time aspect of games typically require very quick manipulation of game state, and it is thus preferred to maintain this data close to the executing game logic. This also means that game state may be lost if a game server crashes. Therefore, some of the game state is occasionally saved using the persistence layer such that it can be recovered, should a game server crash.

Comparing the client layer and the presentation layer, location prediction of objects is often used on the clients to minimise updates sent. In order to do this, the clients must maintain a replica of the virtual world that exists on the game servers.



[Illustration from DAT5 report[6].]

Figure 2.4: A generalised MMO architecture consisting of client, application, and persistence layer.

The generalised MMO architecture consists of three layers, as mentioned previously. This report will disregard the persistence layer and only treat the client and application layer. The persistence layer could be implemented by using off the shelf software.

2.4 Load Distribution by Region Partitioning

This section covers load distribution techniques for MMOGs. This includes a static region partitioning technique known from MMOGs, like: EVE Online. Next, two dynamic partitioning techniques are covered, which are Matrix and JoHNUM. They try to alleviate some of the problems that exists with the static partitioning technique. The main difference between the static and dynamic solutions is the ability to alter the partitioning while the game is running.

All of the techniques chosen uses a geographical region partitioning, in which the virtual world is partitioned into smaller parts, or regions. These have been chosen as interaction in MMOGs in general is geographical bounded, as players only interact with nearby objects.

2.4.1 Static Region Partitioning

When using static region partitioning the virtual world is partitioned into geographical regions, as chosen by a designer and each individual region is assigned to a server. These geographical regions can not be reassigned to another server on runtime. Regions can only be reassigned while the game is stopped, for instance at a daily maintenance time.

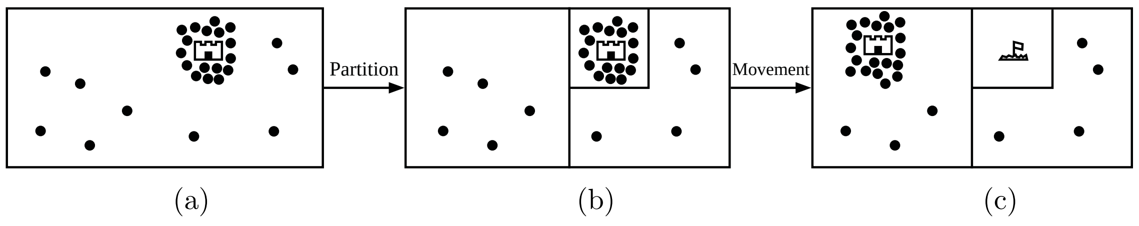


Figure 2.5: Example of partitioning using the static partitioning technique. Black dots are avatars and the castle is a point of interest. (a) Shows avatars flocking around the point of interest, (b) the virtual world partitioned such that the point of interest is in a small region. In (c) the castle has been destroyed and is no longer a point of interest. It illustrates how static region partitioning can give problems in dynamic virtual worlds.

When statically partitioning a virtual world it is worth choosing a partitioning which balances the number of objects in each region, such that the work load will be distributed amongst several regions. It is also with choosing a partitions which minimises the frequency of objects changing regions. This is due to the fact that region changes incur overhead on servers because they must coordinate between themselves to complete the transfer of objects

Further, it is a good idea to choose the region partitioning, such that avatars have as little as possible interaction across region borders, because of the overhead involved with communicating between regions. As well as the overhead involved with having avatars located in the overlap regions, as will be described later in this section.

Achieving a good partition is done by creating the region borders in locations of the virtual world, which objects such as avatars cannot or rarely pass, e.g. building walls, unpassable mountain ranges, and so on. Figure 2.5 (a)-(b) illustrates how a static partitioning could be chosen. The castle illustrates a point of interest and the area around the castle is densely populated. With a well though-out partitioning and a static virtual world, this approach should work quite well, and it is indeed the way many games work today[20].

EVE Online has a natural partitioning of geographical regions in that the virtual world consists of solar systems and spaceships must go through *jump gates* to travel between solar systems. Thus, it seems natural to partition the virtual world into regions corresponding to the solar systems. However, some solar systems experience a much higher population than others[21] and this becomes a problem, because when the server handling the particular region cannot keep up with the work load, as there is no immediate way to remedy the situation.

The solar systems of EVE Online make for natural static regions. Similar natural borders can be imagined for virtual worlds consisting of islands in the sea, rooms of a large building, and so on. The advantages and disadvantages of static region partitioning as opposed to dynamic solutions remains the same. As it is often very hard to predict where avatars will be located at any point in time, determining a good static partitioning can be troublesome. Games with guaranteed predictable movement patterns should generally be well suited for static region partitioning.

However, as games continue to evolve, dynamic virtual worlds become key features. Players increasingly expect to be able to go anywhere and interact with anything in the virtual world. The concept of *free roam*, pioneered by games such as *GTA*, introduced the idea of a virtual world where you can go almost anywhere. Additionally, games are continually pushing the limit to how many objects players can interact with, e.g. move, destroy, and so on. These factors in contemporary games are not in harmony with a statically partitioned virtual world. This is illustrated in Figure 2.5 (c), and as shown in the figure, a point of interest may move. This is especially likely in dynamic virtual worlds as everything can be moved, destroyed, or created anew by players. As the point of interest change or moves, the new location will become densely populated and the initial static partition will become a liability for the performance of the game.

Further, as the size of virtual worlds increase, it becomes infeasible to have the game producers design all of the virtual world. Thus, the concept of user-generated virtual worlds is being applied increasingly. This also fits quite poorly with a statically partitioned virtual world, as it is impossible to predict player movement and interaction patterns in user designed areas of the virtual world. Section 2.4.2 and Section 2.4.3 presents two attempts to deal with these problems.

Overlapping Regions

When an avatar travels between regions (and thus, must be handled by different game servers) in the partitioned virtual world, it will at some point find itself in an overlapping region, that is, the point σ_i in which the object is located has a non-empty consistency set. Overlapping regions are necessary in order to sustain the illusion of global consistency. While an avatar is located in an point σ_i belonging to server S_i, with a consistency set $c(\sigma_i) = S_j$, it will receive information about objects on S_j as well as objects on S_i. Figure 2.6 shows the overlapping regions and an object residing in a point in an overlapping region. The client controlling the avatar located in σ_i will also notify the servers in $c(\sigma_i)$ about its movements, orientation and so on. Without overlapping regions, it could happen that objects would suddenly appear in front of your avatar, seemingly out of nowhere. This could happen every time objects crossed into the region in which your avatar was residing.

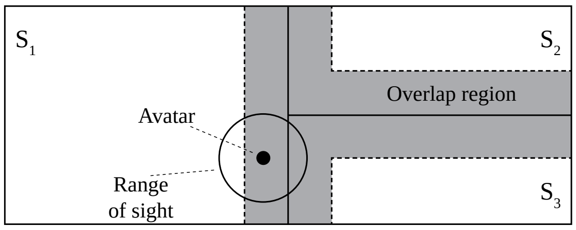


Figure 2.6: Illustration of overlapping regions.

Advantages and Disadvantages

The static partitioning technique for distributing a virtual world onto several servers has its advantages and disadvantages. The advantages are as follows:

- It has been tested in contemporary games.
- There is a high degree of control over which regions of the virtual world is managed by which servers.
- Static region partitioning is relatively simple to implement.

The disadvantages are:

- The inflexibility of the static partitioning technique makes for a two-fold problem
  - Overloaded regions cannot be remedied without taking the virtual world off-line and re-partitioning the virtual world.
  - Similarly, under-loaded regions cannot be increased in size, without taking the virtual world off-line.
- The static partitioning technique needs a designer to tell which regions in the virtual world is managed by which server. This is tedious work for a designer and relies on the designer to have knowledge of possible locations, which might be subject to overload. User-generated virtual worlds further complicates this.

2.4.2 Matrix

Matrix[7] is a dynamic partitioning solution, which uses a simple dynamic partitioning scheme to solve the problem of handling a large number of players in a virtual environment. The partitioning scheme concerns virtual worlds where movement is restricted in a two-dimensional fashion, like in World of Warcraft. Matrix does this by using a number of game servers, matrix servers, and a matrix coordinator to handle the increase in work load, when a region of the virtual world has an increase in population. The game servers, matrix servers, matrix coordinator, and clients are described below and illustrated in Figure 2.7:

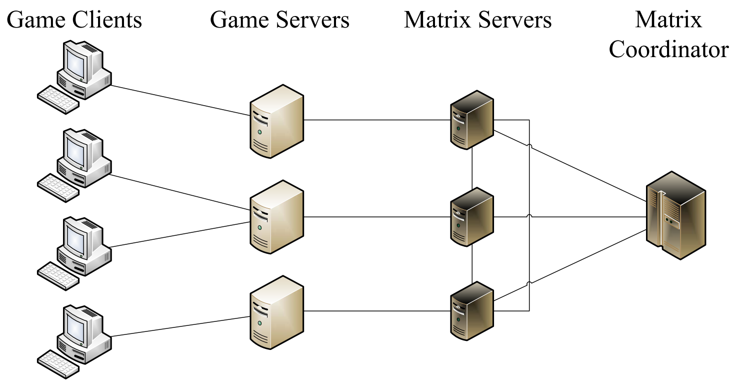


Figure 2.7: Architecture of the Matrix solution showing clients, game servers, matrix servers, and the matrix coordinator.

Game Clients: The game clients are used by the players and provides a visual representation of the content to the players and allows players to interact with the virtual world. The game client is connected to a game server and can dynamically change game servers as it moves in the virtual world.

Game Servers: The game server stores the current game state. It is also responsible to manipulating game state according to player interactions, physics, and game logic.

Matrix Servers: Each game server has a matrix server attached to it. Matrix servers are aware of which region their game server maintains, and which game servers are adjacent to its own region. The game server indiscriminately passes all its messages to the matrix server. The matrix server then relays the messages to the correct matrix servers. The game servers therefore does not need to know anything about other game servers, since this is the priority of the matrix servers. The matrix server is a separate process from the game servers.

Matrix Coordinator: The matrix coordinator is a single instance, which maintains the overlapping regions that is used by the matrix servers. The matrix coordinator only informs the matrix servers about overlapping regions, when a new partition or merge is invoked.

The reason to have the matrix- and game-server separation is to be able to divide the tasks. The game servers tasks are specific to the game and the game rules, as it enforce the game rules, such that the virtual objects conform with these. The matrix servers is used to distribute the virtual objects among the game servers. The matrix servers handles all the tasks concerning distribution. The game servers relay all information from the clients to the matrix servers which then again relay it to the correct game server according to the distribution. This separation of tasks ensures the game servers are separated from the distribution tasks, which is handled by the matrix servers[7].

Partitioning Scheme

To partition a virtual world or a region that a game server holds, Matrix uses a simple “split-to-left” technique, as illustrated in Figure 2.8. This technique creates two geographically equal-sized regions, where the left region is handed to a new game server. This is illustrated in Figure 2.8, where a region is partitioned into two regions, seen after arrow (1). If a region needs to partition again, then it partitions the region on the same dimension as previous as seen after arrow (2).

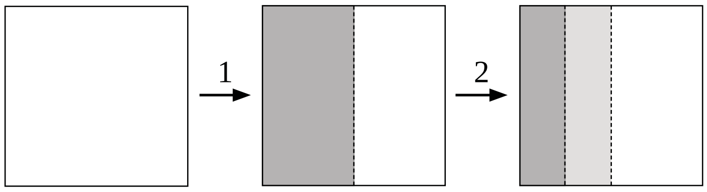


Figure 2.8: Matrix partitioning technique: 2D-”split-to-left”.

An example concerning the use of the “split-to-left” technique and the dynamic solutions, which Matrix is, is illustrated in Figure 2.9. The black dots are players and the castle is a point of interest for all the players. Given that most players are concentrated around the castle it leaves the game server in an overloaded state, as seen in Figure 2.9 (a). The game server can then initialise a partitioning of the virtual world, and share the load with another game server, as seen in Figure 2.9 (b). However since it uses the “split-to-left” technique it requires another partition, since the first partition leaves an almost empty right-part to one of the game servers. If all the players decide to move to another region, then Matrix has the option of merging the two right-most regions, as they are no longer overloaded. Then Matrix can use the freed server to partition the left region, where many players have moved to, as illustrated in Figure 2.9 (c). As it is able to merge and partition regions while the game is running it can react to a moving hotspot. This gives more freedom to the designers of the game as they are not restricted to, e.g. having big battles only at some specific locations, but can have them all over the virtual world.

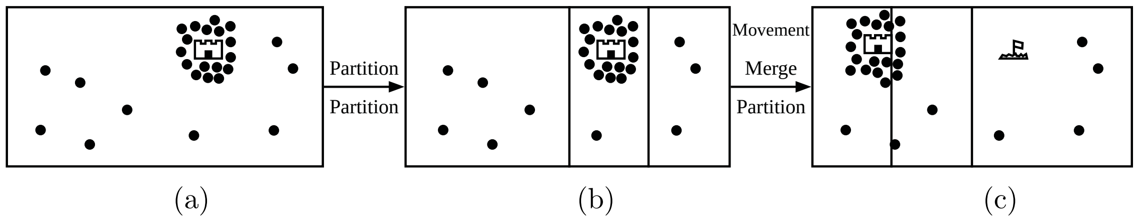


Figure 2.9: Example of a partition with Matrix's "split-to-left" technique.

The decision to either partition or merge is done locally by the matrix server. This decision to partition or merge is based upon the work load of the game server. There are thresholds for when a server should partition and when it should merge, this could e.g. be above 90% CPU usage to partition, and below 20% to merge on both game servers. These thresholds have to be determined according to the hardware and game type, such that a partition can be completed before the game server is overloaded. The merge is done in order to ensure that a game server is not underloaded, as it could then be used to service another region of the virtual world, which is overloaded.

Oscillation behaviour can occur. Oscillation is when, e.g. a decision to merge follows just after a decision to partition was made. To a large degree, avoiding oscillation is a matter of choosing the thresholds for merge/partitioning correctly, such that it is possible to partition regions without going below the merge threshold.

The data structure used in Matrix is illustrated in Figure 2.10. The example given in the figure illustrates wants happens when a partition occurs. The initial state of the tree is seen in *a*, where the virtual world has not yet partitioned. In *b* the region held by S_1 is partitioned into two equalled sized regions, where the initial size of the region is illustrated as a dashed node and that it was previously held by S_1. In *c* the region held by S_2 is partitioned into two equalled sized regions, where the old region size is again illustrated as a dashed node. Another partition has happened in *d*, where S_2 again partitions. This method of keeping track of the partitions are used when a game server should merge with another game server, as the merge rules in Matrix states that a server may only merge with another server it has partitioned from. Meaning, when a node merges it must be a leaf node, and its sibling should also be a leaf node, if this is fulfilled along with the CPU thresholds, then the merge can be done.

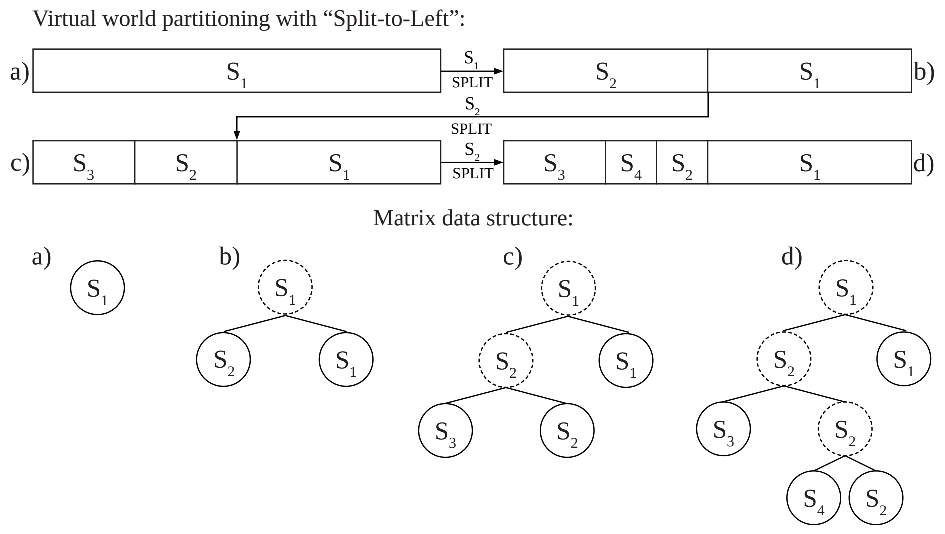


Figure 2.10: Illustration of Matrix's data structure. The tree structure represents the situation after each partition of the virtual world shown in the top of the figure.

Advantages and Disadvantages

The dynamic partitioning technique Matrix has its advantages and disadvantages, especially compared with the static partitioning technique. The advantages are:

- Dynamically partitioning of the virtual world to handle heavy work load in a specific region of the virtual world.
- The dynamic partitioning scheme can also respond to work load being moved around the virtual world, and move server capacity to where it is needed.
- Tries to use a minimum number of game servers to handle the work load.

The disadvantages are:

- The matrix servers is a separate instance to the game servers, which could lead to an increase in latency. This could be built-in with the game servers.
- The "split-to-left" technique does not always yield a good partitioning of the virtual world. This might lead to several more partitions before the load is distributed enough for the game servers to handle.
- The "split-to-left" technique only allows partitioning in one dimension, which can be a problem in large virtual worlds where it could give regions which span the whole width of the virtual world.
- The "split-to-left" technique does not look at the populations in the new areas and can potentially create a region which is empty and one which still has all the players.

2.4.3 JoHNUM

The second dynamic partitioning solution is called JoHNUM[8, 9] and is a solution, which has been inspired by the Matrix solution. JoHNUM uses the same basic idea in its architecture, but has a different approach regarding the data structure, the partitioning scheme, and how the decision of partitioning and merging should be done. JoHNUM is, as Matrix, a technique which concerns partitioning in a virtual world, where movement is restricted in a two-dimensional fashion.

The main focus of JoHNUM has been to perform better partitioning than Matrix. The main difference from Matrix is that JoHNUM uses a collection of techniques to find an even balance for partitioning. This even balance is found by looking at the population distribution in the virtual world and then trying to create a partitioning of the virtual world, where half of the population is in one of the region sets, and the other half is in the other region set. A partition has to live up to a criteria before its valid. The criteria is that *no isolated* regions must occur in a partition. That is, a region is isolated from its set, if it can not be reached in a traversal from the rest of its regions within its set, as illustrated in Figure 2.11 (a)-(d), where the white set has an isolated region, as seen by the traversal. This is different from Matrix, since it used a "split-to-left" technique which did not look at the placement of the avatars and since it only contained two regions, the no isolated regions criteria was automatically fulfilled. Since partitioning a region into two regions, no isolated regions can occur as the two regions will always be adjacent to each other.

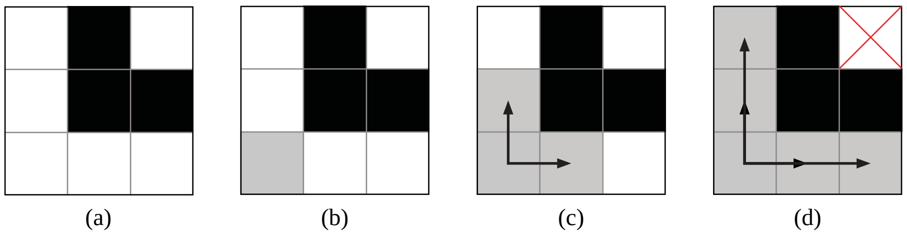


Figure 2.11: Example of isolated regions with two sets, which can be seen with the traversal of the white set (b-d).

In JoHNUM, a game server initiates a partition when the population gets higher than its Maximum Server Capacity (MSC). The MSC is a predefined maximum population limit, which is used to determine if a new partition should be performed. It is desirable to choose the MSC such that partitioning happens before the workload exceeds the amount the server can handle, and in turn, before players have a degraded game experience.

When a partition is commenced, JoHNUM has two possibilities. It is either a partition of a game server, which contains a single region, or it is a partition of a game server which contains more than one region.

In the first case, where a game server has to partition the one region it has, it uses a Region Split Factor (RSF) value to determine how many sub-regions the original should partitioned into. The RSF value can either be 2 or 3. The RSF value is found via a number of steps:

Step 1: Use the RSF value to create a 2×2 grid of the region. Then find the population in each of the smaller regions in the grid. If a region has a population greater than the MSC, then go to step 2 with a 3×3 grid, else go to step 2 with a 2×2 grid.

Step 2: Use the JoHNUM techniques to find a partition, which are explained later in this section. The techniques run until they either find a partition, which is less than a predetermined threshold, or if that is not possible run all techniques and then use the best found partition.

Step 3: The overloaded game server transfers one of the sets of regions found in step 2 to a new game server.

In the second case the game server has more than one region from a previous partitioning. In other words, the game server is handling an area compounded from more than one region. The game server was assigned some number of regions in the last partition, now it must perform a new partition, to further alleviate its work load. The game server uses the partitioning techniques to find a partition, as in the first case. However, as it does only have a subset of the regions from the previous partition, it has to use padding on the regions that it does not control. Those regions are given a population of zero. Then the partitioning techniques are used as in step 2 with the RSF value from the previous partitioning, and before the actual partitioning is made the regions which were padded are removed. However, the new partitions also have to meet the criteria for successful partitions, that is no isolated regions must be in the partitions.

When regions have been partitioned, the overlap regions have to be updated. The overlap scheme used is the same as in Matrix and in turn the same as in the static partitioning solution.

Partitioning Techniques

JoHNUM uses four partitioning techniques to find an even balance in an overloaded region. This is done by partitioning the regions into two non-empty sets, designated s_1 and s_2. Initially, s_1 will contain all the regions allocated to that particular game server. Some regions will be taken out of s_1 and put into s_2. After the two sets are found, s_1 will be assigned to the overloaded game server, and s_2 is assigned to a new game server. As an example, if the game server handling s_2 is overloaded, it will move some regions to s_3 and s_3 will then be assigned to a new game server.

There are four techniques, which are used when partitioning. The first two are used, when the RSF value is 2 or 3, whereas all four are used when the RSF has a value of 3. This is because, that the last two techniques does not give any new unique combinations, when the RSF value is 2. The four techniques are illustrated in Figure 2.12 (a) – (d) and described below:

Left-to-Right (LR): Scans the regions in a left-to-right motion, adding a region to set s_2 and removing it from s_1 as it moves across the regions. This is illustrated in Figure 2.12 (a), where it starts from the top-leftmost region scanning to the right, then continuing to the next line of regions, when it has added the rightmost region. It does not add the last region to s_2, as illustrated in the figure, since this would make s_1 empty.

Top-to-Bottom (TB): Does the same as the left-to-right technique, but does it in a top-to-bottom motion instead, as illustrated in Figure 2.12 (b).

LR-and-TB: This technique does a left-to-right and top-to-bottom scan, as illustrated in Figure 2.12 (c).

LR-TB-with-Diagonal: The last technique does the same as LR-and-TB, but it also adds the diagonal, as illustrated in Figure 2.12 (d).

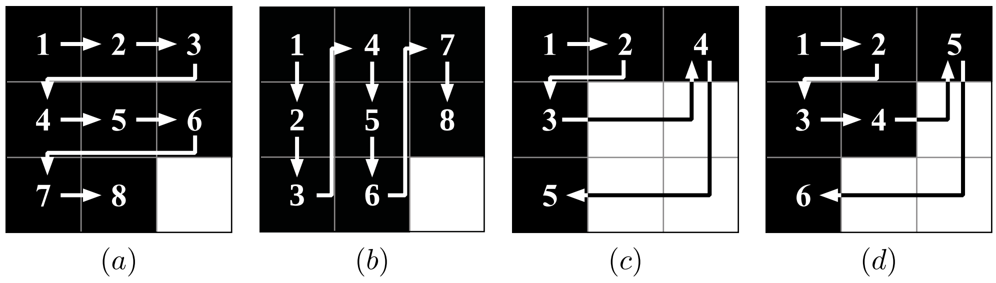


Figure 2.12: Techniques for finding good partitions of a region in the virtual world. Regions are scanned to determine which assignment of regions into sets s_1 and s_2 gives a good distribution of players. The techniques are: (a) Left-to-Right, (b) Top-to-Bottom, (c) Left-to-Right and Top-to-Bottom, and (d) Left-to-Right, Top-to-Bottom with diagonal.

If any of the four above techniques reach a condition in which the total population of the regions in s_2 is greater than the total population of the regions in s_1, it will halt. Because the population of s_2 can only increase as the algorithm progresses, the distribution will only become less equal. There is therefore no reason to continue.

Thresholds for how equal the distribution of players must be, are set. If at any point, the difference in population drops below this threshold, the algorithm is also halted. This is to avoid examining too many cases, because the found assignment of s_1 and s_2 is good enough according to the thresholds that has been assigned.

These partitioning techniques have been selected, such that isolated regions do not occur, as it would increase the number of times a player could switch servers, when moving around in the virtual world. To get as many unique combinations as possible and still maintain the criteria of no isolated regions, the partitioning techniques are used from two corner cases:

1. Starting in top-leftmost, as illustrated in Figure 2.12 (a)-(d).

2. Starting in top-rightmost corner.

The total number of unique combinations, which conform with the non-isolated regions is 32 for the 3×3 and 6 for the 2×2[8].

In order to keep track of which partitions can be merged, it is necessary to keep track of which partitions have been made. For this a tree data structure, similar to the one used in Matrix, is used. An example of the data structure can be seen in Figure 2.13, where the initial state of the tree can be seen in a. When the region held by S_1 is partitioned it creates a dashed node with the size of the initial region, and creates two child nodes for each of the game servers. Each child node contains the regions held by a server. Further partitions can be seen in c and d, where a white node indicates a region that is required to be partitioned into sub-regions, if it is overloaded. As JoHNUM partitions the tree created is going to be more broad than in the case of Matrix's data structure, where a partition always has two child nodes.

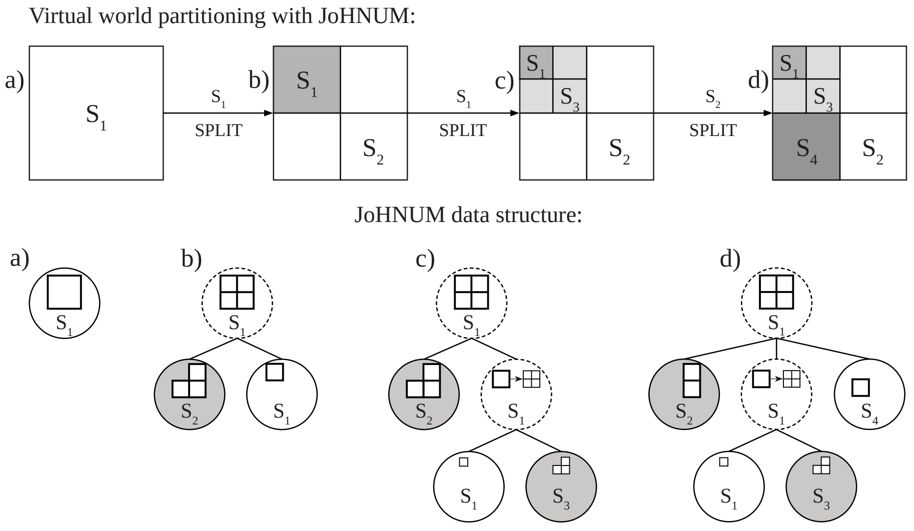


Figure 2.13: Illustration the data structure used in JoHNUM to keep track of which partitions has been made. The data structure illustrates wants happens, when the virtual world is partitioned.

However JoHNUM does not include a description of how merging of regions occurs, but the authors mention that a merge of regions between two servers should only occur, if they are on the same level in the tree. The merge should also still satisfy the criteria of no isolated regions, because if two game servers merge and an isolated region is created, then a later partitioning of the regions might not be possible, since the partitioning of regions must apply to this criteria. This has been confirmed via email with one of the authors, which can be read in Appendix B.

In Figure 2.14, an example of how an actual virtual world would be partitioned is depicted. The initial load at (a) is partitioned into a 2 × 2 grid, where the top-right region is handed to a new game server, as shown in (b). Then in (c) the load has moved to a new region, and JoHNUM would then merge the top-right into the large area and partition the top-left region again, as it would be overloaded.

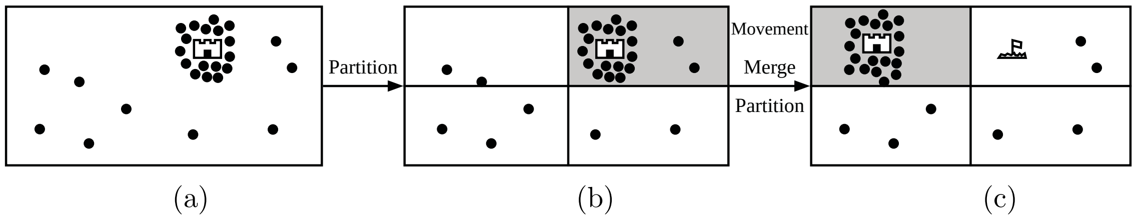


Figure 2.14: Example of a partitioning with JoHNUM techniques.

Advantages and Disadvantages

JoHNUM has its advantages and disadvantages, as Matrix had. However since JoHNUM is inspired by Matrix and therefore has examined various ways of improving the partitioning of a virtual world, it does have some significant advantages:

- JoHNUM has, like Matrix, the advantage of dynamic partitioning of a virtual world and can therefore dynamically allocate server resources to stabilise the work load.
- JoHNUMs partitioning techniques are not locked to one dimension as opposed to Matrix, which gives equal-sized regions, instead of slices.
- JoHNUM tries to balance the population as much as possible, when partitioning the virtual world it takes distribution of players in the virtual world into account, by looking at a large number of smaller partitionings, 2×2 and 3×3 partitions.

The disadvantages are:

- The partitioning techniques are much more complicated than Matrix, where Matrix used a simple "split-to-left" technique.
- The partitioning techniques can potentially create a partitioning, where one region contains all the players and another which is empty, which would give no improvement in performance after the partition. This can only occur if all players are located in the same region.

2.4.4 Optimal Solution

It could be interesting to investigate the optimal solution in order to use it as a benchmark for the previously mentioned solutions.

The optimal solution with respect to scalability will have a linear relationship between the number of players it can support and the number of game servers supporting these players. In essence, this means every game server will be able to support a constant number of players. During our DAT5-project[6] we tested a single server solution for Rock Pounder. The tests found that a single server could handle about 240 players before exhausting CPU resources, as illustrated in Figure 2.15 (a). We found, that there is a second degree polynomial relation between CPU usage and the number of players. We concluded that this was because of the collision detection algorithm used, which has a time complexity of $O(m \cdot n)$, where m is the number of avatars and n the number of projectiles. The number of projectiles is approximately proportional to the number of players, thus: $O(m \cdot n), n = c \cdot m \Leftrightarrow O(m \cdot c \cdot m) = O(m^2)$.

The relationship between number of players and number of servers in an optimal solution is shown in Figure 2.15 (b). In the study of the optimal solution, there are some assumptions to be made.

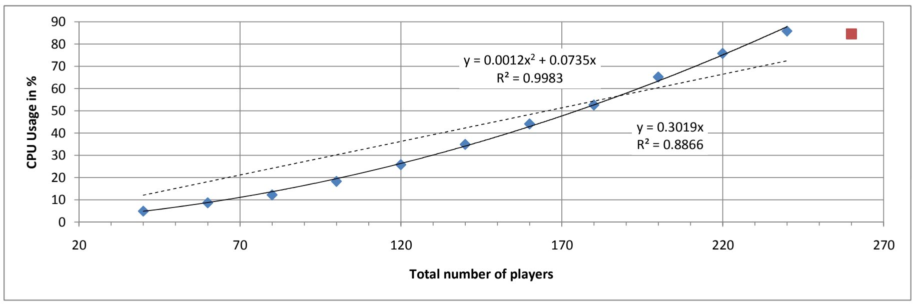
When adding a game server to the pool of servers maintaining the virtual world, there are two options:

- Make the virtual world larger. In practice it can happen by opening up a previously closed region of the virtual world. Then, assign the newly created/opened region to the new game server. This will not change the size of existing partitions.

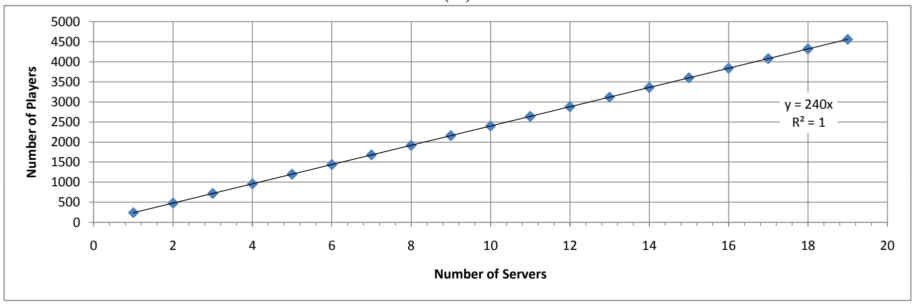
- Keep the size of the virtual world constant. The new game server is assigned an existing region of the virtual world. This means existing servers will have to *give up* some of their regions, decreasing their regions' sizes.

In practice, changing the virtual world size is rarely a viable option. It will seem unnatural for players to experience regions being removed or added in a seemingly random fashion. Thus, this analysis assumes that the virtual world size is kept constant. This also means that adding a server will inevitably decrease the size of existing partitions. In order to be able to reason about the optimal solution, we also assume a uniform distribution of load across the virtual world.

An optimal region partitioning scheme will have no overhead associated with overlapping regions. Further, it will allow avatars to move between regions without overhead. This will give linear scalability. As mentioned previously, our single-server prototype sustained 240 players at once, thus, with linear scalability, the graph should be similar to Figure 2.15 (b).



(a)



(b)

Figure 2.15: (a) The player versus CPU graph from our DAT5-project[6], page 49, and (b) an optimal graph for multi-server given a maximum of 240 players per server as obtained from (a).

However, in order to maintain the localised consistency premise, servers must coordinate in order to give players the possibility of looking into nearby regions and moving between regions. This coordination is inevitably associated with some overhead. This overhead exists only for objects in the overlapping regions. When objects move out of overlapping regions they are only a concern for the particular server maintaining the region in which they reside.

The size of the overlap regions is constant (determined by the range-of-sight for avatars or object size). As servers are added, regions become smaller, effectively increasing the overlap to non-overlap ratio. As such it is not possible to devise a region partitioning scheme which is completely optimal under the premises presented here.

2.5 Summary

This section gave a description of a typical game and introduced the game loop as well as the concept of game state. Then we learned what is expected of a MMOG and the design goals related to developing MMOGs. We found that providing localised consistency is necessary when implementing load balancing by way of virtual world partitioning. Overlapping regions were shown to be help maintain localised consistency.

This was followed by an analysis of the different network architectures available, and the advantages and disadvantages of these were discussed. It was clear that a client/server architecture is preferred over peer-to-peer and hybrid architectures due to trust, control and NAT issues. It was also made clear that this report will not consider problems related to the persistency layer of the network architecture.

Finally, an analysis of three existing load partitioning techniques, namely: Static partitioning, Matrix, and JoHNUM was given. Each technique was detailed, to show how the architecture is structured and how the partition of the virtual world is performed. We showed how static region partitioning may not be applicable in games with unpredictable movement patterns or user-designed virtual worlds. The "split-to-left" technique used in Matrix and the JoHNUM techniques were explained. The "split-to-left" technique seemed rather naive compared to the JoHNUM technique due to the fact that Matrix just splits any area into two equally sized regions without taking the population distribution into account. The optimal solution was presented in order to show the upper bound for scalability. The optimal solution would scale linearly with the number of servers.

Chapter 3

Design of Load Distribution in MMOGs

The design chapter covers our architecture for MMOGs and the design decisions made for the implementation we developed in the process. The architecture is described as a modified version of the Matrix architecture and it is explained which parts of the architecture has which responsibilities.

Subsequently, the particular partitioning of the virtual world we have chosen for the static region partitioning technique is detailed and motivated. The two dynamic partitioning solutions Matrix and JoHNUM are extended such that they can be used in a three-dimensional virtual world. The data structures necessary for the implementation of these techniques, with the our extensions, are presented, and a special extension to the Matrix technique is shown. This extensions actually simplifies the design, and should even improve performance in some cases. The details regarding when the techniques can perform new partitions and when they can merge existing partitions are given along with the explanation of how precisely this is done.

3.1 Architecture

The architecture we have chosen to use for our setup is a modified Matrix design. In this section a description of the differences between the original Matrix design and our extended design will be described. The architecture we have chosen is utilised for all partitioning techniques to have a similar testbeds.

The original Matrix architecture consists of clients, game servers, matrix servers, and a coordinator. This overall architecture has been slightly modified, as the matrix servers are removed and the responsibility has been moved to other servers, as explained in the following:

Game Server: In our modified Matrix design the game servers has the following responsibilities:

Overlap Regions: Maintains their own overlap regions, which is done by using the area of the virtual world that it maintains and by knowing in which directions it has a neighbour. In the original Matrix solution this was maintained by the matrix coordinator.

- **Load Monitoring:** Maintains local load monitoring and information about which areas players are residing in, which is used by the matrix coordinator to choose when and how to partition.

Matrix Coordinator: The matrix coordinator, in the modified design, is assigned the following responsibilities:

- **Partition/Merge:** Maintains the partitioning and merging of the virtual world, which was previously maintained by the matrix servers.
- **Neighbours:** Maintains the different game servers neighbours to be used when creating overlap regions on the game servers. The neighbours are send to the game servers, which then can create their own overlap regions.
- **Partitioning technique:** The partitioning technique is used on the coordinator to find a suitable partition. The partitioning techniques used are either a static partitioning, extended Matrix, or extended JoHNUM, which are explained in later sections.

Our extensions differs from the original design, but it still uses the key concepts of the original Matrix solution. The partitioning of a region should however have been merged with the game servers, but have in this implementation been merged with the matrix coordinator, because of the ability to also support other partitioning techniques. The rest is left for the game servers, which is to maintain their own overlapping regions and player redistribution when partitioning and merging. Previously the overlapping regions were maintained by the matrix coordinator.

The routing of packages also differs in our modified version compared to the original Matrix design. We have chosen to have the possibility that clients can be connected to multiple servers at once. This occurs when a client has a visibility range that spans into two regions or more, then the client can receive information from several regions from multiple servers. In the original Matrix design this was maintained by the matrix server, such that the game server got the information from another game server via the matrix servers, and then relayed it to the client. However in our design the number of hops the messages should go through are reduced, and this is the reason for choosing this modification.

When the clients are connected to multiple game servers, it is important that the information is not sent twice, as some objects in the overlap regions are at the same edge of two regions. To remove this problem regions only sends information about objects they control, and not replicated objects. Replicated objects are objects that are controlled by another region, but has a visual range within another region. The other region then creates a replicated object. This ensures that the object information is only sent once, as only one server at the time can control an object.

In addition to this the overlap region in our extended Matrix design can be smaller than in the original Matrix design, because the regions does not need to have information about the client when it can see into another region. This is because the region that has the client tells the client to connect to the other region, which will then receive information about that region. The region is however not required to do any game logic on the client, since it does not reside within it. The only information another region needs is to be able to perform the collision detection and it is only needed, when the hull of the object crosses the region border. This means the overlap region can be as small as the hull of the object.

The clocks on the clients and servers are synchronised, because they are used when sending messages between each other. The servers sends messages which can be about movement of objects, positions. These positions are send over a network with latency. The client then needs to predict the position of the object with the time on the message it was sent. It does this by extrapolating the position given and the time it takes sending the message. The prediction is used on clients and servers, where the servers predicts the error from the correct position to the predicted. If a prediction is incorrect, then the server can send a new position to the client, such that it can predict a more accurate position.

3.2 Static Region Partitioning

The static partitioning technique to partition a virtual world onto several game servers has been successfully utilised in several popular games, like EVE Online[1], as mentioned in Section 2.4.1. The static partitioning has in our solution been utilised in a three-dimensional environment, where the partitions created are cubes.

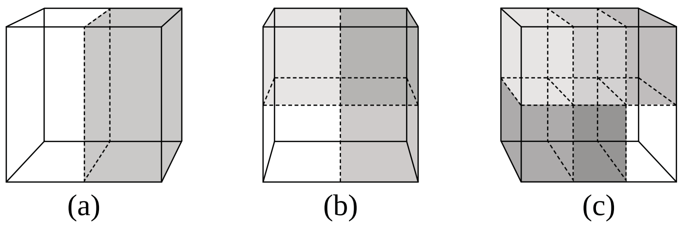


Figure 3.1: Illustration of the static region partitioning.

Three partitionings of the virtual world has been chosen, as depicted in Figure 3.1. The virtual world is an open space cubic world with no walls or other obstacles, which can be used for natural partitioning. In addition, if the general movement of the objects in the virtual world is not known, e.g. unpredictable moving hotspots, then the best distribution which we can create is to partition the virtual world into equal-sized regions. The partition of the virtual world for two servers is depicted in Figure 3.1 (a), and in (b) and (c) the partitions for four and six servers respectively is illustrated. The reason for multiple partitionings are to be able to test with two, four, and six game servers. In a real example all the servers available would have a region of the virtual world allocated, as the partition can not be altered when the game is started.

The static partition has also been chosen to use overlap regions, as stated in Section 2.4.1, to be able to provide localised consistency.

3.3 Matrix

The original Matrix solution was described in Section 2.4.2, which is a dynamic approach to achieve load distribution in a virtual world. The architecture used for our extended Matrix solution was described in Section 3.1.

Papers concerning the Matrix “split-to-left” technique describes only how it is applied in a two-dimensional virtual environment, or in a three-dimensional virtual world, where movement is constrained in a two-dimensional fashion. However, some games take place in a three-dimensional virtual world, where movement is not constrained to only two dimensions. Rock Pounder is such a game, and the MMO game: EVE Online[1] is also such a game. It is

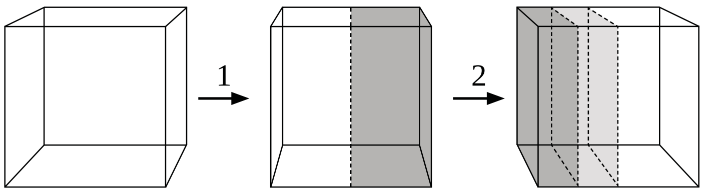


Figure 3.2: Our proposed three-dimensional "split-to-left" technique used with our extended Matrix solution.

therefore necessary for us to design an extended version of the Matrix technique, such that it can be used in a three-dimensional environment, but still be *true* to the original design. The method chosen to extend "split-to-left" is depicted in Figure 3.2, and is expanded into three dimensions via using slices of the virtual world, such that it is a plane which partitions the two new regions, instead of a line as in the two-dimensional version of the "split-to-left" technique.

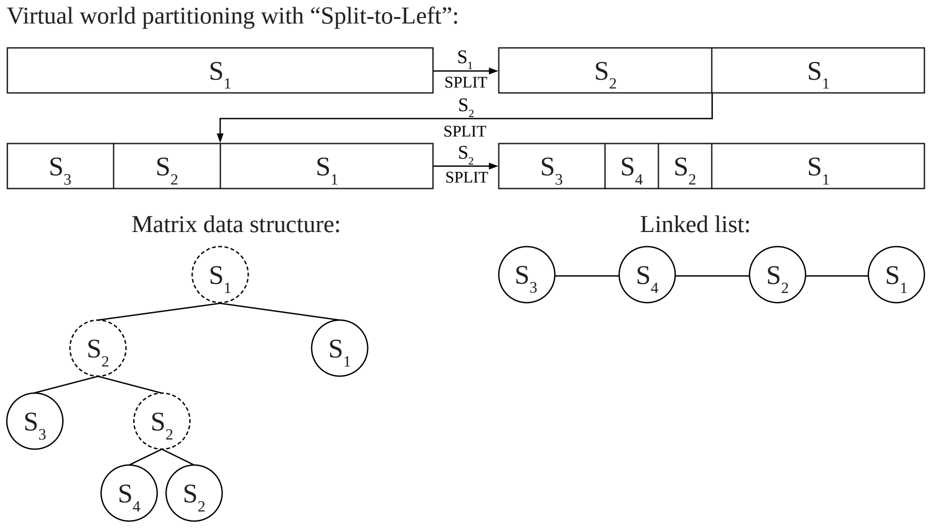


Figure 3.3: Illustration of Matrix's data structure and our proposed chained list data structure.

The last modification is the data structure, that the game server and matrix coordinator maintains; both regarding the regions, but also the neighbour list. The original Matrix design used a tree-like data structure, as illustrated in Figure 3.3 and previously mentioned in Section 2.4.2. This data structure maintains which game server has which region and which partitionings have occurred to create the region. When a partition occurs the two new regions are added as child nodes to the region which was partitioned. As only the leaf nodes are active regions, the intermediate regions are only used to be able to keep track of which child regions can merge into their original region, their parent. This data structure is constructed, such that the structure can be partitioned and merged in the same order as they were created, as was described in Section 2.4.2. However, we propose a different data structure, which is comparable to a linked list, as illustrated in Figure 3.3. The left-most game server, S_3 has a neighbour called S_4. S_4 has both S_3 and another game server S_2 as neighbours, and so on. This modification was made, such that the game servers could partition and merge with neighbouring servers and thereby not need to look to children or parents to either partition or merge.

The reason for this new structure is to avoid having two adjacent regions, which are underloaded, but cannot merge. An example of this can be seen in Figure 3.3, where S_4 and S_3 in the last partition cannot merge in the tree structure, whereas it is possible in the linked list approach. To have S_4 and S_3 merge when using the tree structure, S_3 and S_2 would have to merge before this can happen, and if S_2 is overloaded then this would not be possible.

3.4 JoHNUM

The original JoHNUM solution is a modified edition of Matrix, as mentioned in Section 2.4.3. The modifications lies in the use of a new data structure and partitioning/merging techniques. Instead of using the "split-to-left" technique for partitioning it uses a series of techniques to partition regions. This is done by either creating a 2×2 or 3×3 pattern of sub-regions, which is then distributed across two game servers. However, when an overloaded server is responsible for a set of more than one sub-region, then JoHNUM redistributes the sub-regions that it contains, instead of creating new sub-regions. JoHNUM therefore only partitions regions into a 2×2 or 3×3 pattern of sub-regions if it contains a single sub-region and is overloaded.

The original JoHNUM solution is only designed for two dimensions, but for our game these techniques are required to work in three dimensions. For this purpose, we have extended the original JoHNUM techniques to work in three dimensions. In our extended version of JoHNUM four partitioning techniques are still utilised in conjunction to find a partition of the virtual world. To extend the techniques to include the third dimension, and still be true to the original design, we have utilised the same constraints to find a partition, as the original JoHNUM has used in its four partitioning techniques.

These four partitioning techniques conform with the three following criteria for finding valid subsets, when partitioning. Where the transport criteria is added as an extra criteria, but is also mentioned in the original JoHNUM, but not defined as a criteria:

Non-Empty: When partitioning, there must be no empty set of sub-regions. Which is a sensible constraint, as a partition is made to lessen the work load. Therefore both game servers must receive regions to control, and not just one of them.

Isolation: No sub-region should be isolated, when a partition is created into two sets of sub-regions. A region is isolated if it is not possible to make a path from the region to all other regions in the same set of sub-regions. This path is to be made in such a manner that it is only possible to go from one region to another if they are neighbours.

Transport: Moving from across, either horizontal, vertical, or depth, from one sub-region to another, there must not be movement from a sub-region from the set S_1 to a sub-region in set S_2 to a sub-region in set S_1, as illustrated in Figure 3.4 (a-c).

Partitioning Techniques

We have extended the four partitioning techniques used in the original JoHNUM design to be able to handle three-dimensional partitionings. The details of the extensions are explained

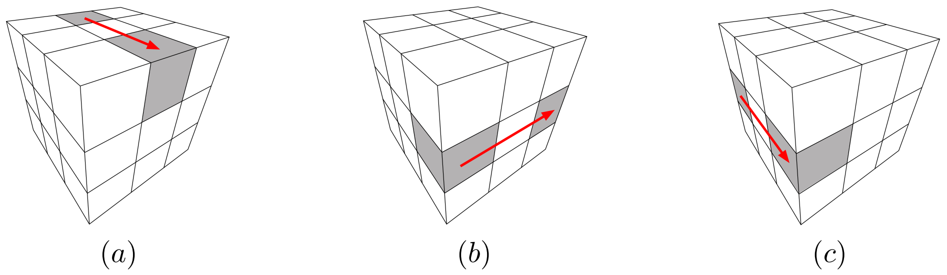


Figure 3.4: The transport restriction, where movement is (a) horizontal, (b) vertical, and (c) depth.

in the following. The four partitioning techniques are used in combination and yields a number of unique combinations. These all maintain the property of no isolated sub-regions, as the original JoHNUM and therefore should minimise the number of server changes needed. While these four techniques do not find all unique combinations, they do find a small subset of the unique combinations, which maintains the previous mentioned criteria. The number of unique combinations chosen to check have to be weighted against the time it would take to perform all the tests. As it is expected to run in a realtime environment it is desired to be able to quickly decide which partition to use and therefore only a small number of unique combinations are examined. In addition to this, the combinations examined should be unique, e.g. only one set of unique combinations from the top-left corner and one set from the top-right corner. Therefore, as with the original JoHNUM design our extended techniques will use similar techniques with different start locations.

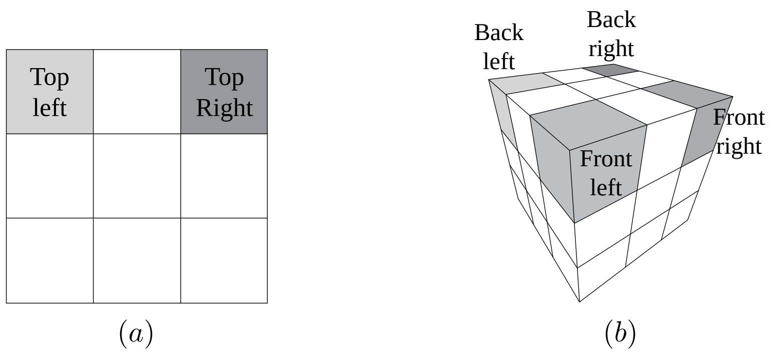


Figure 3.5: JoHNUM scanning combinations with the four techniques: (a) Two dimensions (two corners) and (b) three dimensions (four corners).

It increases the number of unique combinations in the set by using two corners of the pattern, which is illustrated in Figure 3.5 (a). However given that the game engine is in a three-dimensional virtual world, where movement on all three dimensions are possible, it is necessary to partition the region in three dimensions, instead of just two and repeat the techniques with four corners, as illustrated in Figure 3.5 (b). This creates a bigger difference in the number of found unique combinations that maintains the property of isolated regions. However, such a difference is acceptable given that the four extended partitioning techniques yields an acceptable number of combinations. The four extended techniques have the following additions:

LRRows: Examines in a left-to-right, row-by-row, and *layer-by-layer* fashion.

TBColumns: Examines top-to-bottom, column-by-column, and *layer-by-layer*.

LRaTB: Examines left-to-right, top-to-bottom, and *layer-by-layer*.

LRTBwDR: Examines left-to-right, top-to-bottom, *layer-by-layer*, and with the diagonal.

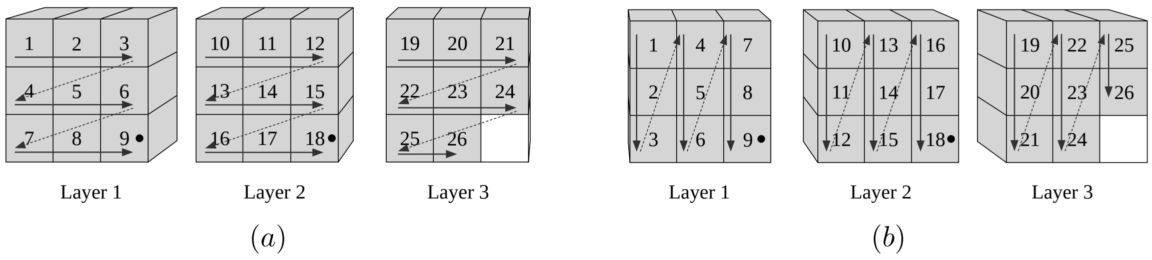


Figure 3.6: The partitionings techniques: (a) LRRows and (b) TBColumns.

All four partitioning techniques have been added the layer-by-layer search. However, they all use it in a different manner. The *LRRows* basically examines the combinations in the same order as the original. However, the original design would have stopped examining combinations, when getting to the last of the nine sub-regions, as illustrated in Figure 3.6 (a), layer 1. The extended will proceed with layer 2, and finally layer 3. The *TBColumns* examines in the same manner, first examining layer 1, then layer 2, and finally layer 3, as illustrated in Figure 3.6 (b).

The first two partitioning techniques are very similar to their original counterparts, however the last two techniques are a bit trickier than their original counterparts, since the addition of the third dimension makes the examination a bit more varied. The last two techniques are however only used when the partition is $3 \times 3 \times 3$, whereas the first two techniques are used in $2 \times 2 \times 2$ and $3 \times 3 \times 3$.

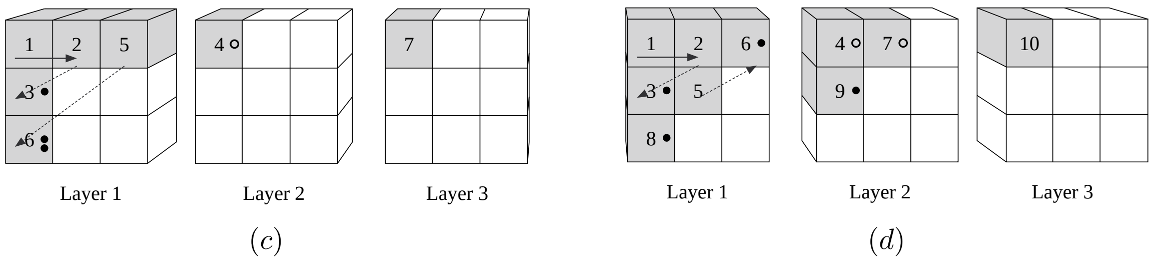


Figure 3.7: The partitioning techniques: (c) LRaTB and (d) LRTBwDR.

The *LRaTB* examines the sub-regions by moving right, followed by moving down from the initial starting location, and then by moving to the next layer from the initial starting location. This is illustrated in Figure 3.7 (c), where filled dots represents moving up one layer ($1 \to 2$), and empty dots represents moving down one layer ($2 \to 1$). This is done to get a set of sub-regions that are always in close proximity to each other or clustered together.

The *LRTBwDR* takes this even further, using the same approach as the extended *LRaTB*, but including the diagonal. The result is a staircase scenario, as illustrated in Figure 3.7 (d).

When partitioning a game server with multiple regions, it only contains a subset of the regions and must therefore use padding with regions it does not contain. The padded regions are set to have a zero population, as in the original JoHNUM design. However we have added an additional check to ensure the criteria of isolation and transport is verified, as it could be possible to create a partition, that does not satisfy the specified criteria, when the padded regions are added. An example of this can be seen in Figure 3.8 (a)-(b), where (a) illustrates that two regions can be merged, even though they are not adjacent and therefore at the situation of a merge invalidates the no-isolated region criteria, but with a partition its valid again. However in (b) no partition can ever be applied, since there exist three regions that does not satisfy the no isolated regions criteria. A partition will therefore always have a subset of regions, which have isolated regions. It has therefore been decided to add the additional check to verify the criteria, when merging and partitioning regions.

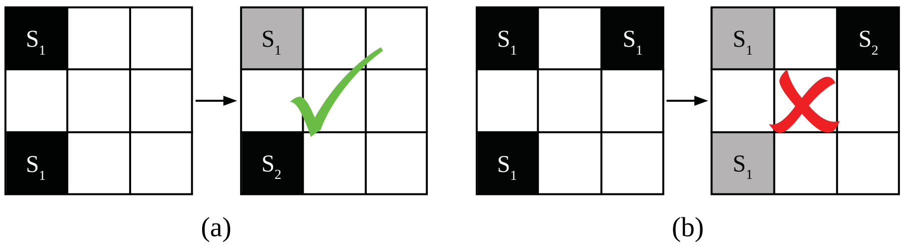


Figure 3.8: Illustrations of two possible scenarios when merging, (a) valid partition and (b) where no valid partition is possible.

Structure

When a partition is made a tree-like structure is maintained, as described in Section 2.4.3. However, we have also made an extension, as the list of neighbours for each region is also stored, as can be seen in Figure 3.9. This addition is done to be able to merge, which is explained later in this section.

When to Partition

In the original JoHNUM the decision to partition relied on the MSC, however we have decided to rely on CPU usage of the game servers, instead. A game server will partition, when the CPU usage is above a given threshold. This is done as more than the players are a factor in Rock Pounder, e.g. projectiles. Therefore we believe it to be a good choice to use CPU usage is a good metric for work load, as a game server with a low number of players can be overloaded if it contains a high number of projectiles. In addition to this the MSC value has to be chosen to fit the hardware's overload limit, whereas the CPU usage already takes this into account.

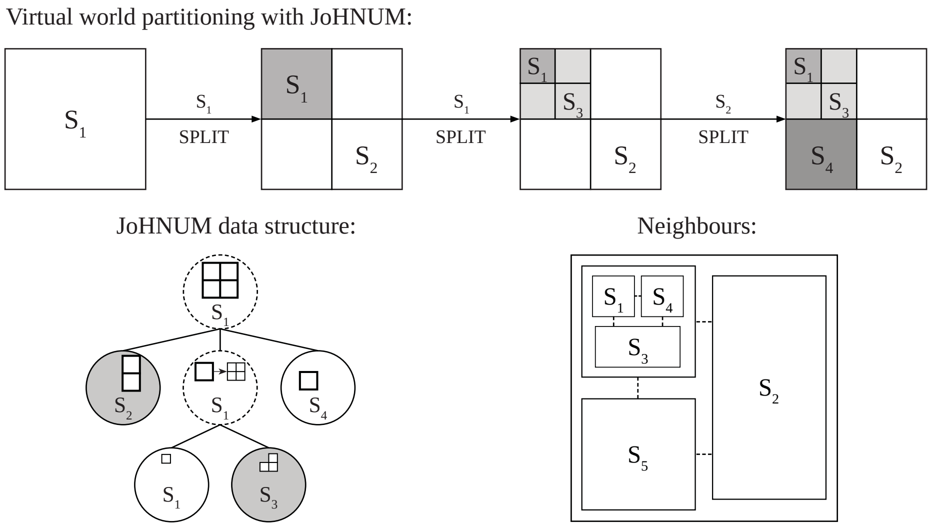


Figure 3.9: Illustration of JoHNUM's data structure and our added neighbour-list for merging.

When a partition is to be created a RSF value is chosen, which is either 2 or 3. In the original JoHNUM the value was chosen based upon the MSC value as described in Section 2.4.3. However in our extended design, both RSF values, 2 and 3 are examined to find the best partitioning. This was decided, because we wanted the most balanced partitioning.

Furthermore, when a partition is selected the region-set with the lowest population is transferred to the new server. This is done to minimise the transfer of objects between the servers.

Merge

In the original JoHNUM design, it was not specified how merge should be handled. For this reason we have devised a method, such that merge can be implemented. Before two game servers can be merged, their regions have to conform with these rules:

1. The two game servers region sets have to be siblings in the tree-structure.

2. The two game servers region sets have to be adjacent, which means at least one of the regions in one set has a neighbour in the other set. This can be ensured by using the previous mentioned structure, where the neighbours are stored when a partition is created.

3. The combined set from the two servers has to conform with the previous mentioned criteria of isolation and transport. This is needed as these criteria are tested when a partition is to be made, and if a merged region does not conform with these criteria, it can end in a state which cannot be partitioned again.

In addition, if a game server contains all the sub-regions from a partition, all the sub-regions are removed and merged into the larger parent region, which then again can merge with the regions that are siblings in the tree-structure.

Simplified

The extended JoHNUM techniques and the addition of a dimension can possibly yield a large amount of combinations, so a simplified version could be devised to alleviate this problem. This approach could instead use a extended version of the "split-to-left" technique, that Matrix uses. Instead of doing a one-dimensional partitioning of the sub-regions and only doing it half-way, the following extensions could be made and still use the approach of $2 \times 2 \times 2$ and $3 \times 3 \times 3$ cubes:

Dimensions: Allow partitioning on all three dimensions, instead of just one (like "split-to-left").

Sub-regions: Allow partitioning on a layer, a row, or a column.

These two extensions creates three very similar techniques for partitioning the virtual world, that are: Row-by-row, column-by-column, and layer-by-layer. The three techniques are illustrated in Figure 3.7 (a) Row-by-row, (b) column-by-column, and (c) layer-by-layer.

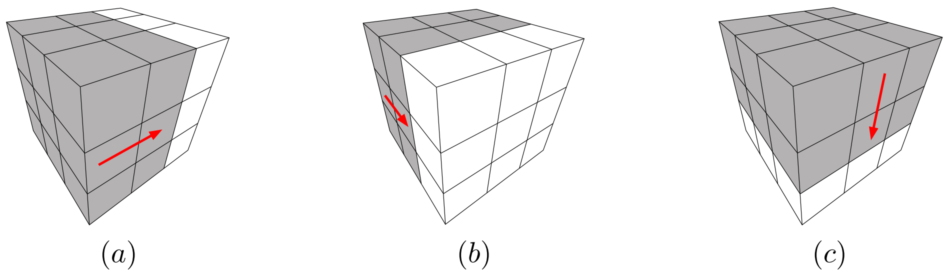


Figure 3.10: The simplified partitioning techniques: (a) Row-by-row, (b) Column-by-column, and (c) layer-by-layer.

Comparison

The difference between the original JoHNUM techniques and the extended versions is not only the addition of a third dimension, but the number of combinations that the algorithms can find. The extended JoHNUM techniques gives a total of 256 unique combinations out of 59,188 possible that also conform with the criteria, which is 0.433% of total. The number of combinations were found via a tool that we wrote, which can be found on the appertaining DVD in Appendix C. The number of combinations the extended techniques can find can quickly be searched with modern computers, so a real-time system like a game should not be affected. So the simplified techniques six unique combinations with the $3 \times 3 \times 3$ and three unique combinations in the $2 \times 2 \times 2$ is not needed, since we can use all 256 unique combinations from the extended JoHNUM techniques and the simplified is already a subset of the extended JoHNUM techniques.

The difference in the number of unique combinations found by the techniques and the total number of unique combinations is very big. This is because the small number of unique combinations needs to be valid with all of the criteria: Non-empty, isolation, and transport. The reason for this test is when a game server has more than one region in its region set, then it has to use padding, which can create isolated regions in the combinations.

As the partition should be used in a realtime environment the time which it should take to create the partition should be minimised. However there still has to be enough unique combinations to be able to create a good partition.

We have chosen the 256 unique combinations, such that they represent different partitions, which should be able to partition the work load in different locations in the virtual world. Therefore the selected combinations are similar, but with a different starting point as shown in Figure 3.5 (b). There will be no testing of the simplified, because of the before mentioned reasons.

The extended JoHNUM solution has its advantages and disadvantages against Matrix, which are:

Advantages:

- More fine grained partitioning of the sub-regions, than the extended “split-to-left”.
- Higher chance of even distribution of players, than using the extended“split-to-left”.
- Uses player locations to create an even distribution.

Disadvantages:

- More complex than the extended “split-to-left”, both regarding technique complexity, but also neighbour complexity of the regions.
- More computably intensive, than the extended “split-to-left”.

3.5 Summary

This chapter covered the network architecture created to support a multi-server solution and thereby a MMOG. The network architecture is inspired by the Matrix architecture, so it covers the client, game servers, and a coordinator. The clients gives a visual representation to the players and takes input from the player and sends it to the game servers, which does the appropriate actions with the players input and sends information about the objects positions back to the clients or other game servers. The coordinator maintains the respawn of avatars and sends them to the correct server that holds the respawn area. The coordinator has an additional responsibility with the dynamic approaches, which is to find a possible partitioning or merging a game server if it is overloaded or underloaded.

Next the static region partitioning was covered. It introduced how partitioning of a three-dimensional virtual world could be achieved, where movement in all three dimensions were applicable. Then Matrix, which described the extensions created, such that it could handle three-dimensional partitionings. It also described the modified data structure, where the original Matrix uses a three-like data structure, it was emphasized that a linked list was more appropriate for the extended Matrix design, since it allowed a more flexible merging. Lastly an extended JoHNUM design was created, which covered creating four techniques that would apply to three dimensions, and the increase in the number of combinations the extra dimension would add. The worry about the increase in combinations was however not justified, since the 256 unique combinations was feasible to apply to a realtime application, like a game.

Chapter **4**

The Game

This chapter describes the game developed with this report for the purpose of evaluating load distribution techniques in MMOGs. First, the details of the general game engine we have developed will be given. This is followed by details specific to the game developed on top of the general engine. Finally, we discuss why the game is applicable for evaluating MMO load distribution techniques for games in general, and not just for Rock Pounder.

4.1 Game Engine

This section gives an overview of technical features of the game engine, that we have created for Rock Pounder. The primary focus will be on the architecture, communication, and potential performance issues. This section therefore contains a structural overview of the game engine to clarify the chosen engine architecture. Afterwards, features of the collision detection and what limitations it is subjected to, will be described. Furthermore, the measures taken in the network layer to avoid large clustering of sent messages, which could lead to breakdowns are described. Lastly, details of how the engine handles the distribution of the virtual world are given along with a description of how this affects the various parts of the engine. This section pertains to the game engine used in Rock Pounder. However, the game engine is constructed in such a manner that the game on top of the engine could be substituted with any three-dimensional game, where the player controls an avatar in a MMOG environment.

4.1.1 Network Architecture

The game suite consists of three parts: Clients, servers, and a coordinator, as depicted in Figure 4.1. The connections between the three parts are highlighted. In the following sections each of the components are described and their responsibilities outlined.

Client

Each client controls an avatar in the virtual world and is connected to servers. A client can be connected to multiple servers. This is the case if the client's avatar can see into a region held by another server, than the server it currently is on.

The client is responsible for giving a visual representation of the virtual world to the person playing the game. It is also responsible for handling input from mouse and keyboard,

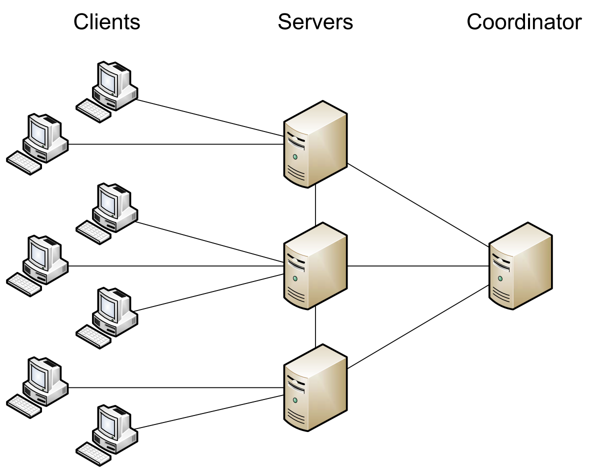


Figure 4.1: Overview of the layout of clients, servers, and the coordinator.

which is transformed into actions that will be sent to the server. This allows players to control avatars in the game.

To test with hundreds of players or avatars within the virtual world, a test client is needed. This is due to the difficulty of gathering hundreds of human players for testing, when it is needed. A test client must be able to send actions to servers according to the information available to it, and input similar to what a human player would do when playing the game. A test client also gives the benefit of not needing a graphical representation or handle keyboard or mouse input, as a client used by human players needs. These elements have been removed to make the test client more performant. This performance is needed as a large number of test clients are desired in the tests and because the amount of hardware available for testing is limited.

Server

Servers are responsible for maintaining the virtual world. This includes enforcing the rules of the game, e.g. if a player is hit by a projectile the damage is subtracted from the players health. The server is responsible for all the game logic related actions in the virtual world like, when people can shoot, how they can move, if projectiles have hit an avatar, and so on.

There are multiple servers, which each have been assigned a region of the virtual world. Servers communicate amongst each other to maintain the virtual world as a whole. Servers are responsible for transferring objects between them, if e.g. an avatar moves from a region assigned to server A, to another region assigned to server B, then server A must transfer the object to server B. The details of this transfer is explained later in this section. Before a server can transfer an object to another server it must have a connection to the server. This connection is created to the neighbours of the server. A server knows which servers are its neighbours from the coordinator in the dynamic approaches, or from a static file in the static partitioning approach.

Servers are also responsible for sending information about the virtual world to the clients and handle the interactions made by the clients. The information sent to the clients is predominantly object position updates, however, clients also receive information pertaining to server changes, and generally all information relevant for the client in order to be able to play the game.

Distribution: To partition the virtual world onto several servers and allowing interaction across the servers, we chose to create a geographical partitioning of the virtual world and create regions. Partitioning into regions allows avatars to move between regions.

When a server controls a region, and not all of the virtual world, then its load is diminished, because only the objects in the servers region have to be considered with the game logic. In addition only the messages from the avatars inside the region have to be handled, and the server only has to send messages to the avatars which are in inside and in the range of sight of the region. However on the bordering edges to other regions, some extra information is needed, e.g. for the collision detection, as will be described later. Before collision detection can be done the servers must replicate the objects on the region borders. The number of objects on the borders is expected to be low compared to the total number of objects. Thereby a speedup in a few components is possible, because of the distribution of regions onto multiple servers. This is under the premise that it is possible to create a geographical partitioning, which partitions the objects between the servers.

Changing Region: The distribution of the virtual world on to several servers with geographical partitioning requires that an object can change region. This was described in Section 2.4.1 as overlapping regions and why they are necessary in such a partitioning. However, it was our experience that this simple model could be improved upon. Thus, an object crosses several borders before it reaches the actual *region* border. These borders can be seen in Figure 4.2 and what they are responsible for are explained here:

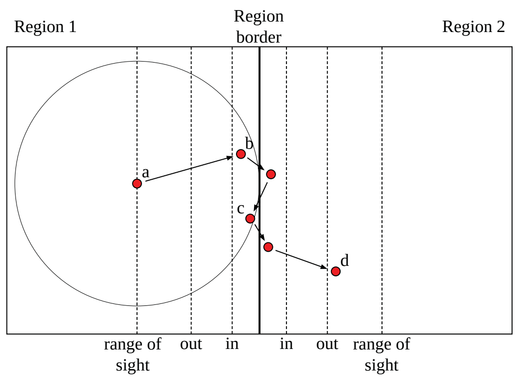


Figure 4.2: Illustration of the three borders: Range of sight, out, and in. An object (a-d) moves across the different borders. Region 1 is handled by Server 1 and Region 2 is handled by Server 2.

Range of Sight: The range of sight border is responsible for determining when an avatar is able to get information from the other server. In Figure 4.2 at location a, an avatar has moved from inside region 1 towards the *region* border. When the range of sight border is reached the avatar begins to receive information from region 2, as the avatar can see into the other region. The distance from the range of sight border to the *region* border is typically set to be the distance an avatar can see. That is, the range of sight of an avatar.

In and Out: When the *in* border is crossed, the object has moved from inside region 1 towards the *region* border, the object is then replicated onto region 2 as it needs to apply collision detection and be able to interact with the object. If the object moves outside the *in* border again it keeps being replicated onto region 2, until it passes the *out* border. The two borders are used to avoid rapid switching between being replicated and not replicated.

Region Border: When the *region* border is crossed from region 1 to region 2, control of the object is transferred from region 1 to region 2. This only needs to transfer control as the object is already replicated onto region 2, as the *in* border has been passed.

In Figure 4.2 an avatars traversal through the borders can be seen in the locations (*a*-*d*). At location *a* the avatar can see into region 2 and starts receiving information from this region. At *b* the avatar has moved across the *in* border, and is now replicated onto region 2, which can test if any part of the avatar is over the region and collide with any objects in region 2. Then the avatar moves over the *region* border and back again, this transfers the control firstly to region 2 which then updates the replica in region 1. Shortly after, when the avatar moves back over the *region* border to *c*, region 1 assumes control again and now updates the replica on region 2. Lastly the avatar moves to *d*, which means it changes region again, and when it reaches *d* it is outside the *out* border and the replica of the avatar is removed from region 1.

In order to minimise unnecessary region changes, a time limit has been introduced. That is, after a server change, an object waits a minimum time before it will change servers again. This is illustrated in Figure 4.2 - object *b* to *c*. However, when the time limit has expired, and the object has crossed the border to a new region, it will change to the new server immediately.

Coordinator

The coordinator is connected to all the servers in the system, and there is only one instance of the coordinator running. The coordinator is responsible for coordinating *respawn* of avatars, among other things. Respawn occurs when an avatar has just connected and entered the virtual world or has died from an impact with a projectile. When a player's avatar is respawned from dying, the avatar can, e.g. be reset with full health, and be moved to its starting location to be able to go into a new fight and try to shoot avatars from the opposing team again. It is the coordinators responsibility to ensure the avatar information gets to the server it should respawn on. This is because the respawn area might reside in a region, that is not adjacent to the region the avatar it currently resides in. Another reason for having the coordinator handle respawn is to be able to globally determine the spawn location of the avatar.

The dynamic solutions give the coordinator additional responsibility, namely, determining which servers should handle which regions of the virtual world. The decisions of when a partition or merge should be made is also the responsibility of the coordinator. In addition to this, when a change is made in the partition the coordinator pushes information about the new partition (sizes and locations) to the affected servers. In the static partitioning solution, the coordinator, however, does not need to have this responsibility, as the regions sizes and locations are always known.

When to Partition and When to Merge: When the ability to partition and merge exists in the engine, there is still an issue of when to create the actual partition or merge. The desired effect of the partition is to ease the load of the server when it reaches a critical level. This critical level is set to be before the game experience is disrupted. Lots of factors impact the load, e.g. number of players, number of projectiles, bandwidth used, and so on. To look at all these factors and determine when to partition would require a large number of rules, however all of these factors affect the CPU usage of the server. The bandwidth, however, only affects the CPU usage lightly. In these types of games this is, however, not a problem. Therefore a simplified parameter to look at is the CPU usage to determine when the server should partition. To choose exactly when to partition a threshold has to be set, so when the CPU usage gets above the threshold, a new partition should be made. This threshold has to be chosen in a manner, which creates a partition before the server reaches an overloaded state.

Besides partitioning, servers also has the ability to merge. This is done to be able to utilise resources better, e.g., if two servers use almost no CPU, then they can be merged and the freed server can be used to service another region of the virtual world which is overloaded. This is done via a threshold, like with the partitioning, however in this case it has to have CPU usage below the threshold on both servers before they can merge. The merging is still subject to the restrictions of the individual techniques.

4.1.2 Base Architecture

The game engine is a two threaded engine. The primary thread, the *game thread* is responsible for all the primary game functions, like handling messages, game logic, collision detection, and so on. The other thread, the *communication thread* is responsible for transferring messages to and from the coordinator, servers, and clients. This structure allows for the game to receive messages in the communication thread while the game thread is doing something else, e.g. checking collision between objects. In addition to this, transmission of messages is done in the communication thread to avoid delaying the game thread, when waiting for the OS and network hardware to be ready to send messages.

The communication thread is constructed in a manner that, when it receives a message it parses and inserts it in a queue. The queue is synchronised with the game thread at the beginning of each frame. The sent messages are also queued in the game thread and when the game thread is at the end of a frame, the messages are dispatched to the communication thread which sends them, asynchronously.

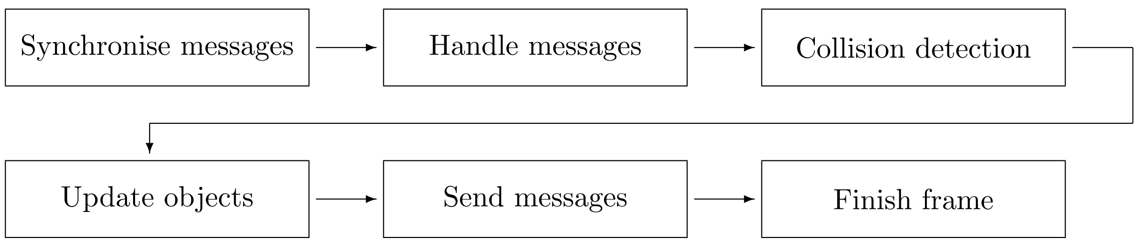


Figure 4.3: The game loop structure, which is used in the servers game thread.

The game thread is constructed, as illustrated in Figure 4.3. It has the game loop which is desired to run at a specific interval, e.g. 30 frames per second. When all components in the game thread have executed, an entire frame has been completed and the engine begins a new frame. The game loop is executed according to the arrows in Figure 4.3 and begins with: *Synchronise messages*. Each component in the game loop is described with tasks and ordering in the following sections.

Synchronise and Handle Messages

The first two components in the game loop is the synchronisation and handling of messages from the communication thread. This could be a login message, which creates a new object and is needed for the avatar, that is controlled by the player. It could also be messages from another server, where control of an object has switched server, or messages from the coordinator giving an update about how the region layout should look like. All of these different messages are dispatched, such that the remaining calculations in this frame will use the latest updated game state.

These components exist both on the client, server, and coordinator as they all have a need to handle messages. The messages are handled differently by the components on the server, client, and coordinator, as they can expect different kinds of messages. For example, clients do not receive messages regarding control of an object, as this is maintained by the servers.

Collision Detection

The third component in the game loop detects if any objects are colliding within the virtual world, e.g. if an avatar is hit by a projectile. A collision is determined by using an avatar's and a projectiles collision spheres[22]. When a collision is found, the proper actions are taken, e.g. if a player is hit by a projectile, then the player receives a given amount of damage from the projectile and the projectile is set to be removed at the end of the frame. The projectile can not immediately be removed as it could have collided with multiple players at once and therefore might be needed in other calculations.

In Rock Pounder a vital part of the gameplay is to determine if an avatar has been hit by a projectile. However, this is also a regular feature to use in other games and is not a feature specific for Rock Pounder. The collision detection is constructed in a manner which is not too computational expensive to be able to perform in a real time environment, such as a game. The tradeoff, for enhancing the speed of the collision detection is the accuracy of object collisions. The collision detection uses discreet time units and only performs collision checks at these discreet times and ignores the places the objects have been between two time units. This method conforms well with the structure of the game engine, where the game state is calculated in these discreet time units, called frames.

The before mentioned accuracy problem in the collision detection, arises when the speed of the objects or the time between the frames is to great, or if the object size is too small. In Figure 4.4 it can be seen how increasing the time between frames, reducing the frame rate, affects the collision detection.

In Figure 4.4 at time t_0 the objects a and b are heading towards each other. Then one frame later at $t_1 = t_0 + \frac{1}{30}$ with 30 frames per second the objects collision can be detected as a is inside b. Similarly at 20 frames per second, with $t_1 = t_0 + \frac{1}{20}$, the collision is also detected, but the objects a and b have moved further than at 30 frames per second. However, at $t_1 = t_0 + \frac{1}{10}$ with 10 frames per second a will not be detected to have hit b, as the time where they have intersected has passed and a is on the opposite side of b. In Figure 4.4 the objects went through each others centre or middle of the objects, but if a and b intersect at

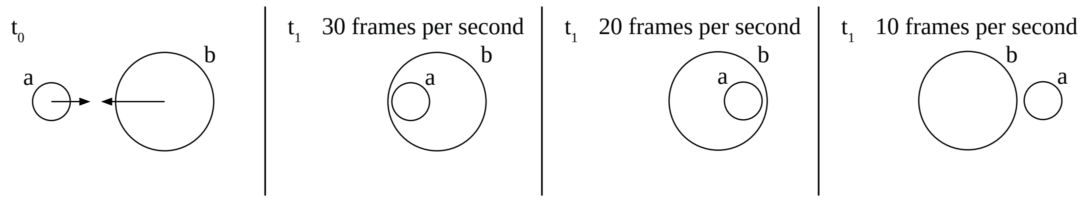


Figure 4.4: How collision detection are affected by frame rate, where collision is detected at 30 *and* 20 *frames per second, but not at* 10 *frames per second at time unit* t_1*.*

each others periphery, then the frame rate needs to be higher to detect the collision. For this reason the frame rate, speed, and size of the objects has to be selected so the desirable effect of the collision detection is achieved. The amount of imprecision accepted is determined by how the feeling of the game should be, so it conforms with the game rules.

The frame rate therefore has to be above a threshold for the game to behave properly according to the design of the game. To have Rock Pounder conform with the game design, we have chosen that Rock Pounder should always have a frame rate above 25 frames per second. At this frame rate the game has a smooth feeling, which conforms with the game design.

The collision detection component is only handled by servers, as clients can not be trusted to decide their own collisions. This responsibility is therefore given to the servers, which can take the appropriate actions when two objects collide.

Update Objects

The fourth component is responsible for updating the positions of all the objects in the virtual environment in accordance with velocities - and the results from the collision detection.

On the servers, the real position of the objects are calculated, which later in the game loop are sent to the clients. The servers have the correct position of an object at all times, so clients does not calculate their own positions, because clients can not be trusted with this task. Servers also checks if the new position of an object is outside the regions the server controls, and if it is, it sends a transfer control message to the server, which should assume control of the object.

The client also calculates a position, but can not reliably calculate the correct position, because of network delay between the client and server. Therefore the client will predict the position via dead reckoning[23], according to the last updates it has received from the server. The client uses these extrapolated positions as the true positions, which are displayed as the objects positions for the player. Another reason to use prediction is to avoid sending messages at a high rate to give the feeling of a fluent game, as prediction provides a fluent game feeling, instead of jumping objects, because of inadequate number of updates.

The prediction is also executed on the server to compare the real position with the position that the clients predicted, and if the difference between these two is too great, then the client is expected to have calculated a wrong position and is updated with the correct one. The time it takes before the prediction is wrong depends on the actions the player performs. In addition to this, if the server makes a bad prediction it will only send an additional message to the clients, which will not disrupt the gameplay as the clients already expect updates with a frequent interval.

The coordinator does not have objects, which needs updates so this is not used on the coordinator.

Send Messages

During all the previously described components, the messages which has to be sent were queued. In this component they are combined into larger packages which are sent to the receivers. This is done to reduce overhead of sending a lot of messages, as ten small packets would have much more overhead than one large packet, because of headers on each sent packet.

The majority of messages sent from the servers are information about the whereabouts of the other objects in the virtual world. The game engine uses extrapolation on the clients, which predicts the possible positions of the objects it receives information about. This prediction is also performed on the server, such that the server can compare the correct position of an object with the predicted position and only send an update, when the prediction is inaccurate, as mentioned before. This greatly reduces the amount of messages required, but it is a tradeoff for precision of the objects on the clients.

In a game the load is not always evenly distributed. There can be certain times where all avatars will take the action simultaneous, e.g. if a game mode has a location an avatar should be in to get points (like king of the hill) and when the point (or hill) moves, all players will start to move to the new location. This kind of sudden peak load has to be handled, gracefully. The solution chosen in the game engine is to drop updates about objects if the load is to great, as these are sent with a regular interval, so if a single or few messages of this type is dropped it will not disrupt the game play. However, if these are continuously dropped, the game play will be disrupted as the player will not receive information about the other nearby objects. The alternative to dropping messages is to make the server wait and block for the time it takes to send all the updates at once. However, if this takes too long the game play will also be heavily disrupted. In this design, dropping packets was opted for, after the wait and block approach failed to provide an acceptable experience for players.

This component is used in all three parts; the client, server, and coordinator, as all have the need to send messages.

Finish Frame

The last part is to cleanup the current frame. This includes removing objects which has been marked for deletion, e.g. the before mentioned projectiles and players. It is safe to remove these objects as we have reached the end of the calculations for this frame, and they are not needed for any further calculations. In addition to this some maintenance tasks are performed to prepare for the next frame, which on the client includes readying the graphics card to draw a new frame.

4.2 Test Game: Rock Pounder

Rock Pounder has been build on top of the general game engine described in the previous section. Rock Pounder is a space shooter set in a three-dimensional virtual world in outer space. The player controls a spaceship (avatar) belonging to either the red or the blue team. The player can move the spaceship freely in all three dimensions and can shoot against the opposing team with a weapon regeneration time of half a second. Projectiles have a lifetime of 20 seconds before they self-destruct. Thus, a player can have a maximum of 40 projectiles in the air at once.

The player is assigned a team, either red or blue when she connects. The goal of the game is to manoeuvre the spaceship and kill opponents by hitting them with projectiles. These projectiles are small relative to the spaceships.

For each projectile which hits a spaceship, the health of the spaceship deteriorates. When a spaceship is hit by the third projectile, it dies. This effectively means that the position of the spaceship is reset to a predetermined area, called a spawning location, and speed is set to zero. Spaceships on red team are reset to a different area, than spaceships from blue team. Further, health is restored.

One very game specific aspect of game development is avatar control. In Rock Pounder players can apply *thrusters* by pressing keys on the keyboard. In effect, a message describing the pressed keys are sent to the server periodically. The pressed keys describe a wish to accelerate. The server then applies this acceleration to the internally maintained velocity of the spaceship. The spaceships position is then updated from the velocity periodically. This gives a space-like feeling when controlling the spaceship.

Similarly, projectile behaviour is very game specific. Projectiles in Rock Pounder are created on the servers. If a player holds down the “shoot”-button on her keyboard, this will make the periodic messages sent from her client include the desire to shoot a projectile. The server receives this and creates a new projectile, if half a second has passed since the last shoot-event by that avatar. As projectiles always travel in a straight line in Rock Pounder, it is only necessary to update clients with information regarding the projectile once. Clients can extrapolate the trajectory of projectiles after they have received the initial projectile information, because it contains the time, location, and velocity at the point of creation. As mentioned previously, clients synchronise clocks when they connect to a server for the first time. While this synchronisation is not completely precise, it is good enough to show the projectile on the clients screens.

4.3 General Testing Applicability

This section will explain why results obtained in evaluation of load distribution techniques using the game: Rock Pounder, are not, in fact, limited to this particular game, but can be generalized to other games, and even other genres.

Rock Pounder can be characterized as a MMOFPS space shooter. Due to the fast-paced game play that is available in Rock Pounder, it has a low tolerance to latency akin to that of FPS games, compared to the higher latency tolerance in commercial MMOGs, such as World of Warcraft and EVE Online. The game can therefore be used as a base measurement for FPS games, both in earth- and space-based virtual worlds. However, a key point is that the partitioning techniques have been extended to support games with three-dimensional movement, whereas most FPS games limit objects to a very small range in the third dimension. This does, however, not limit the test game, since two-dimensional techniques can be applied to the game engine, and the extended designs mentioned in the previous chapter can also be used.

The game specifics, e.g. moving and shooting in Rock Pounder does also apply to other games, especially FPS games, since this is a shooter where close interaction and fast-paced nature is the essence of the game. This is often one of the key aspects of other FPS games, and may therefore also apply to them.

A main cause for worry is the movement patterns in Rock Pounder. The game will create close battles between two teams, which will create natural hotspots within the game. This is also seen from other FPS games, where a sudden burst in interaction and players can be seen in one area of the virtual world. This is often the main worry for game creators, since this is a situation where the game engine and network can be applied the greatest amount of load. Since FPS' have many of these hotspots, and FPS games have the strictest requirements to latency for any genre, FPS games can be considered a kind of worst-case for games. Therefore, Rock Pounder should be an excellent candidate for a test game for evaluation of load distribution techniques. The movement patterns are also seen in other games, like: EVE Online and World of Warcraft, which can have hotspots in certain locations of the virtual world, e.g. cities. So this should also be applicable for other game genres, which has the problem of hotspots.

4.4 Summary

This chapter covered the general game engine developed for Rock Pounder. It covered the network architecture, which will be used with all the extended partitioning techniques. The different responsibilities were covered: How the servers would handle overlapping regions, how transfer of objects work, how prediction was used to minimise bandwidth usage in both the client and server, and so on. It covered the base architecture for the game engine, how the game loop is used to create a frame. The different components were covered in the game loop, e.g. synchronise and handling of messages, collision detection, updating object states, and so on.

Specifically, a problem regarding degrading frame rates and collision detection was emphasised. We argued that 25 frames per second is the limit for when collision detection stops behaving according to players expectations.

Next the test game was described, which is called Rock Pounder. It explained that Rock Pounder is a FPS game where two teams compete in eliminating each other. Lastly, we argued why Rock Pounder is valid as a test game. The reasons is that it requires a low latency, because of the fast-paced nature and that games which do not require low latency should be equally applicable.

It also covered that it had some of the same scenarios, that happens in other MMOGs, which is the sudden creation of hotspots in the virtual world.

Chapter **5**

Comparison of Partitioning Techniques

The previous chapters detailed load balancing solutions, allowing MMOGs to scale with the number of players and handle the movement of players in the virtual world. This chapter describes the various tests conducted to evaluate the of these solutions.

The goal of the tests is to determine how well the different partitioning techniques balance load and how well they scale. This is done by observing how well game servers cope with many players participating in the game. We measure scalability based upon the number of players which can be served, in relation to how many servers are used.

This chapter includes the test environment and metrics used in the tests.

To be able to evaluate the partitioning techniques we have selected two test scenarios which are detailed in this chapter. During the analysis of these test scenarios we decided to further investigate two scenarios which also included in this chapter.

Finally, a section for each of the test cases containing the expected results of the tests, and the actual test results are shown. This is concluded with a comparison between the different techniques.

5.1 Environment

The static and the two dynamic techniques, which were described in Chapter 3 are tested by using *Benedict*, a cluster of computers at Aalborg University. The cluster consists of:

Computers: 35x 2.8GHz Intel Pentium 4 (Northwood), 2GB memory, and 1Gb Ethernet.

OS: Ubuntu Karmic Koala (9.10).

Network: HP ProCurve 2900 48-G switch with 1Gb Full Duplex Ethernet.

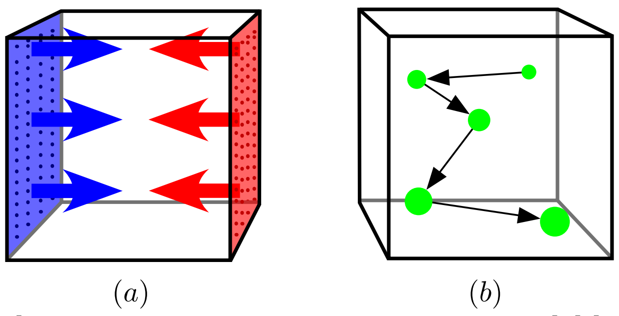
All data is stored on one central network share, which the nodes have access to. This means any log data written to stable storage can be flushed over the network at any given time. As all the test machines share this storage, and if all machines flush their data within a short time period, a saturation of network or storage device bandwidth could occur and block the write operation. This can result in a short term drop in frame rate as the write operation are blocking calls and can block the game loop in the game engine.

Tests were conducted with 10 test client instances running on each machine for the most part. In tests with more than 280 clients, 20 test client instances were used per machine, due to lack of machines. For tests with up to 280 clients, between 1 and 28 machines were used for test clients. For tests with over 280 clients, between 15 and 25 machines were used for test clients. It was determined by the CPU, bandwidth, and frame rate logs that even with 20 test client instances running on each machine, at no point was the CPU or network bandwidth exhausted.

Each server instance ran on its own machine.

5.2 Scenarios

In this section two test scenarios are described. They are both motivated by MMOGs. The test scenarios are also related to the static and dynamic techniques, mentioned in Chapter 3. These two test scenarios are illustrated in Figure 5.1 (a) and (b).



[Illustration from DAT5 report, p.48[6].]

Figure 5.1: Illustrations of two test scenarios: (a) Head-on and (b) hotspot.

The two test scenarios have been chosen for their relevance in player behaviour in current MMOGs, and also because they impose different kinds of stress on the different load distribution techniques, e.g. the static technique might be good in test scenario A, but the dynamic techniques might be better in the test scenario B. Both of the test scenarios have been chosen to run for 900 seconds, or 15 minutes, as this time will allow for the game to start, the servers to settle, and a fight to develop and unfold. This can be seen on the videos on the DVD at the end of the report. The two scenarios are described in detail in the following sections.

5.2.1 Scenario A: Head-on

Scenario A is the classical movie scenario where the two armies are standing in front of each other. When the battle begins, they run towards each other and the fight occurs between them. In Rock Pounder this is implemented by having two spawn areas, one for each team, which is on the opposite sides of the centre of the virtual world. When the game starts, the avatars start at their respective spawn area and move towards the enemy and tries to kill them. If an avatar dies, it is respawned at the spawn area for its team, and can again move towards the enemy to try to kill them.

This fight is static in the sense that the avatars will keep the fight in the area between the two spawning areas. The characteristics of the scenario contribute by testing the following:

Contribution 1: Allows testing of the partitioning techniques under static battle conditions.

Contribution 2: Allows simulation of a movie style battle which could be desired to be supported in real MMOGs. We have observed this kind of battle in, e.g. EVE Online.

5.2.2 Scenario B: Hotspot

Scenario B is a hotspot scenario where all the avatars move to a location and fight for a while, for later to move to a new location and fight there. The location in which the avatars fight keeps moving throughout the virtual world. For example, this could represent a fight between castles where the goal is to have the princess at the teams castle. The fight will then move between the castles as the princess is stolen from one team and moved between the castles. The players spawn as in scenario A and then they have to move to the location where the fight is, before they are allowed to begin the fight. This is done to ensure the avatars actually reach the desired fighting location and not just start fighting between the spawn areas, as in scenario A.

This fight is more dynamic as the location where the avatars are allowed to fight, moves, and the avatars then move with the location to be in the fight. This scenario has the following contribution:

Contribution: Allows testing of the reactivity of the partitioning techniques to see how they can cope with a load which is moving around in the virtual world.

5.2.3 Movement

The two test scenarios must be tested with tens and hundreds of players. It would be difficult to find this many human players, as it would require hundreds of people. So to remedy the need for this number of human players, a test client has been created which simulates a player.

The individual movement of a single test client may not be completely realistic when compared to a human player. Test clients follow a scripted path, and as such their behaviour is deterministic, however, the test clients are designed to move such that they exhibit movement similar to movement observed in MMOGs.

The test client can mimic different behaviours in each of the test scenarios as described below:

Head-on: The test clients spawn in a plane around their starting area, as can be seen in Figure 5.2 (a). When a test client has spawned, it then moves towards the opposing team, Figure 5.2 (b-c). When an opposing player is spotted it will move towards it. The test client will then aim towards the opposing player by using a simple prediction of where the opposing player will be when the projectile gets there.

Hotspot: The test clients again spawns in a plane around their starting area. When a test client has spawned it will move towards a designated area of interest. As the test clients reach the goal area, it will begin aiming and shooting, as in scenario A. It will

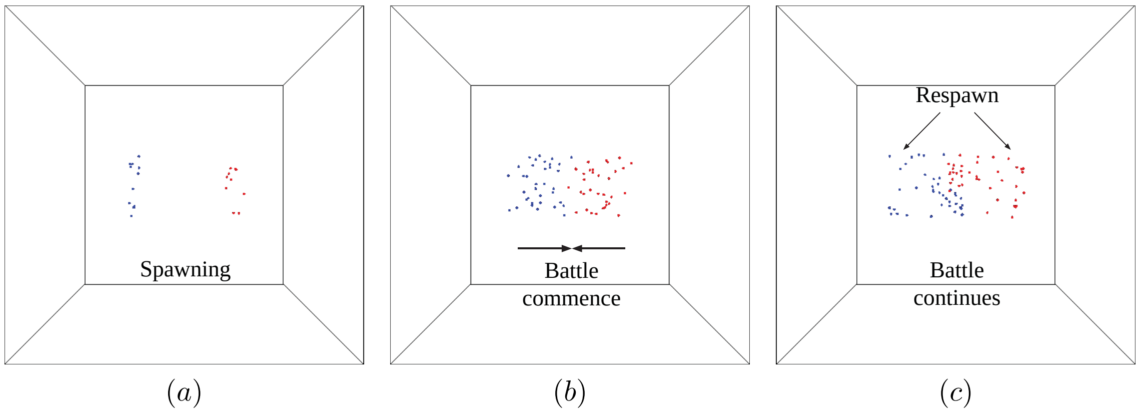


Figure 5.2: Illustration of how test clients move in test scenario A, starting with (a) spawning, (b) start of the battle, and (c) ending with continues battle and respawn.

continue like this until a new point of interest is designated, and the test clients will move towards the new point. Designation of a new point of interest happens for all test clients and at the same time. The location is predetermined by us along with the point in time that it should be activated. They will therefore move in a flock towards a new point, as is illustrated in Figure 5.1 (b).

5.2.4 Static Partitioning

Static region partitioning prescribes partitioning of the virtual world into regions. The chosen virtual world partitioning scheme can have a big impact on the system's ability to handle workloads. Player movement also plays an important role in how well a region partitioned system performs. If players are located close to a region border, it will cause them to be updated about everything going on in the adjacent region. Further, if they are *on* the border, such that they are actually in both regions at the same time, both servers must consider them when performing collision detection. As mentioned previously, collision detection is the main CPU consuming operation in the implementation. Thus, high occurrences of these phenomena can contribute significantly to the work-load on a server. As such, player movement can be a major factor in how well a virtual world partitioning performs.

The virtual world partitions that we have chosen for the static region partitioning in the tests, are illustrated in Figure 5.3.

In each partitioning of the virtual world seen in Figure 5.3 (a-c), each region has been assigned a server. These partitions have been chosen from the perspective that it should be a good partition for the two test scenarios. In Section 3.2 it was mentioned that the static partitioning cannot be altered while the game is running. For this reason it cannot be partitioned to specifically accommodate with the moving load in test scenario B. This is because the load moves from one location of the virtual world to another, and as the load moves, it may be better to change the partition to accommodate this load. The partitioning of the virtual world is therefore best suited by partitioning it into equally sized regions, as the load could occur in any place in the virtual world. This partitioning must therefore both be suitable for test scenario A and B, because both scenarios can happen in a single game.

Because of this we believe the solutions in Figure 5.3 is a good set of solutions, which are suited for both test scenarios.

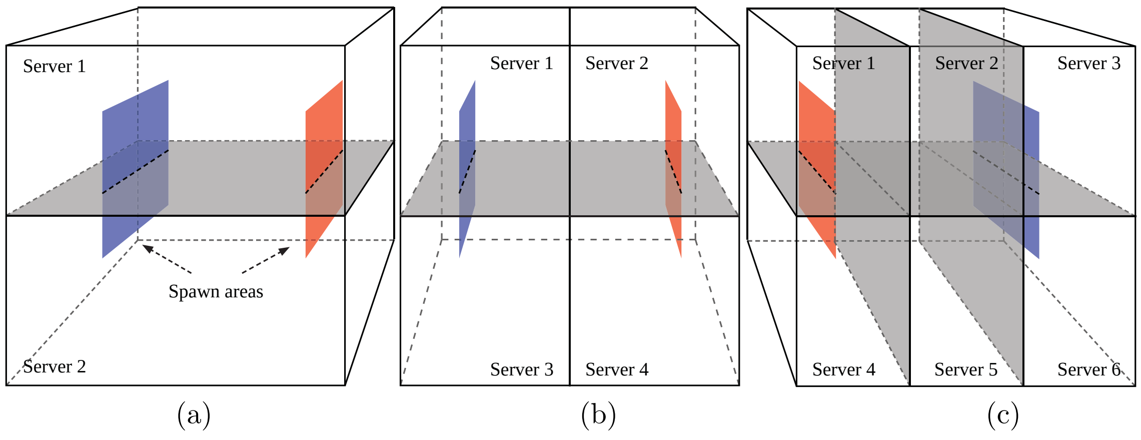


Figure 5.3: Illustrations of the static partitioning with (a) two, (b) four, and (c) six servers respectively. The horizontal plane-segments show starting points for players.

5.3 Metrics

This section covers the test metrics, which we measure to evaluate the performance of the different techniques at given player populations. Presented here is a description of how the test metrics are measured and how the results are compiled into usable graphs.

All of the test metrics are logged to a file once each second. The interval of one second is chosen to be able to see fast alternations which can occur, e.g. when a server is partitioned, and at the same time we do not want to generate too much data. Thus, during the 900 seconds test duration, 900 measurements are recorded to the log file. This is done for both the clients, servers, and the coordinator. The specific metrics which are recorded varies for each.

The game engine has been built such that it gracefully degrades in the face of overload. In practise, this happens by running at a lower frame rate. As the frame rate decreases, the players will start to experience a degrading game experience.

As the primary focus of this evaluation is to determine how many players a specific number of servers can service, it is necessary to determine a frame rate for which the game experience has degraded to such a degree that it is no longer enjoyable. We have decided that if frame rate drops below 25 frames per second it will be regarded as a failure to support that particular number of players. This is because the server has to be above 25 frames per second for the collision detection to work as intended, as mentioned in Chapter 4. However, another part is also the game should run smoothly for the clients, which means the latency should be below 100ms, as mentioned in Chapter 1.

Client

The following describes the measured test metrics on each of the clients. Each metric is sampled and logged every second, for each client connected:

Frame Rate: The frame rate is measured to determine if the client is performing as it should and can handle the events in a timely manner.
Drop in the frame rate is an indicator of too great work load.

CPU Usage: The CPU usage of the client is measured to accompany the frame rate as these two values are tightly connected. The CPU usage can give a view of how loaded a client is as the frame rate is maximum 30 frames per second, and will usually only drop if a resource is exhausted. If the frame rate decreases, but the CPU usage is still low, it can indicate that the I/O bandwidth has been exhausted.

Bandwidth: Bandwidth is measured in sent kbytes/sec and received kbytes/sec. This helps determine if the networks ability to disseminate data has been exhausted.

Latency: Latency is measured with a package sent to the server and back to the client. The time it takes this package to take the round trip is our measurement for latency. Latency can be used to check if messages sent to the server are replied to in a timely manner. This is however both dependent on the load on the network, the client, and the server.

Position: The position (x, y, and z coordinates in the virtual world) of the player, such that the movement behaviour of the test clients can be analysed following the tests.

All of these metrics are as previously mentioned stored into a log file, and sampled each second. We create a log file for each of the clients connected to the game. This is done to be able to use the positions of all the players to be able to do a replay and see the behaviour of the clients, to help analyse the data after the tests.

Server

The test metrics, which are measured on each of the servers is described below along with a description of what the metrics are used for. In addition to the the frame rate, CPU usage, and bandwidth as described under the client, we also measure:

Players: Total number of players in the regions the server controls. This is used to be able to determine the origin of the work load on the server. An example could be if there are many projectiles but a low number of players this would create a low load, however a large number of players and projectiles would create a high load.

Projectiles: Total number of projectiles in the regions the server controls. Which is used in conjunction with the player metric to determine the origin of the work load on the servers.

Dropped UDP: The total number of dropped UDP packets on the server. This is used to determine if a server has saturated its network subsystem and had to drop packets.

The log files from the servers are used as primary measurement for how well the different solutions perform. Frame rate on servers are the key measurement for performance.

The bandwidth to the clients and bandwidth to the servers are measured separately to determine which is the major consumer of bandwidth, between the servers or between the servers and clients. The players and projectiles are measured to determine the number of objects within the region that the game servers holds. The dropped UDP packages are measured to see if the server has reached its maximum capacity for sending packets.

Coordinator

In addition to frame rate, CPU usage, and network bandwidth, the coordinator also logs on the following test metric:

Partition/Merge: When partitions and merges occur it is logged. Region size and total number of regions is also logged. This log is used to be able to visualise how the regions have been partitioned afterwards.

The partition/merge information is however, only logged when a partition or merge is performed. The region data is logged along with a time stamp such that it is possible to synchronise with the other logged data, when analysing or debugging.

Frame Rate Data

When all of the data is stored, there are 900 data points from each server which must be analysed to determine how well the servers performed; if only one server performs poorly, the whole game does not perform as expected. Each executed test has a fixed number of players inserted into the game at the start of the test.

For each test, all the frame rate data for the servers are combined into a single value. This value is found by taking the minimum value of all the frame rate data points, to find the value where one of the servers performed worst during the test. This single value then represent how good a test performed with, e.g. 100 players. All of these values are then combined into a single graph which shows the connection between the number of players and performance, by showing the minimum frame rate recorded. These combined graphs are used throughout the rest of the test section to show the results of the tests.

5.4 Test Results: Scenario A

This section covers the test results gathered by using test scenario A, which was explained in Section 5.2. The results gathered covers a single server solution, and the multi-server solutions: Static, Matrix, and JoHNUM, which will be compared against each other.

The single server solution is only used to define the capacity of the optimal approach, which was described in Section 2.4.4, where it was emphasised that an optimal solution is when the population limit scales linearly with the number of servers. We have therefore tested how many players a single server can handle before it reaches its maximum capacity and begins to drop below a frame rate of 25 frames per second.

5.4.1 Optimal

We tested the single single server solution with test scenario A and is used as the maximum number of players a single server can handle. It was tested from 10 to 280 players, as illustrated in Figure 5.4, where it reached its maximum capacity at 48 players. The number of players which is supported by two, four, and six server can be seen in Table 5.1, assuming the optimal multi-server solution scales linearly with the number of players,

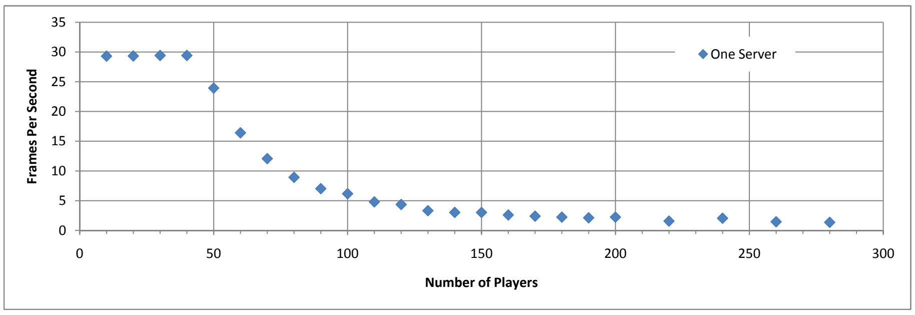


Figure 5.4: Illustration of a single server handling players from 10 to 280 players in scenario A.

Type	One server	Two servers	Four servers	Six servers
Optimal	48	96*	192*	288*

*Table 5.1: The maximum number of players for one server, which has been linearly extrapolated to the optimal solution for two, four, and six servers, for test scenario A. * Extrapolated data.*

5.4.2 Static Region Partitioning

This section covers the static region partitioning technique, which is tested with scenario A. This section therefore concerns the expected results. Then an evaluation of what actually happened is presented. This is compared with the expected result and against the optimal solution, which was mentioned before.

Expected Results

We expect the static technique to perform well with a specific number of players, but not as many as the optimal solution, due to overhead. However, to which degree this will affect the number of players that two, four, and six servers can maintain is uncertain.

Latency and bandwidth are expected to perform well up to and beyond the place where the frame rate drops below 25 frames per second. It is also expected to be the CPU usage in form of collision detection, which is the bottleneck that will reached first.

Results

The static partitionign technique was tested with the number of players from 10 to 280, where 10 to 200 was with an interval of 10, and $200 - 280$ was with an interval of 20. This was tested with two, four, and six servers.

Frame Rate Results: The combined results with the three server configurations and frame rates can be seen in Figure 5.5. Two servers drops below 25 frames per second at approximately 75 players mark, four servers drops below at 127 players, and six servers drops below at 196 players.

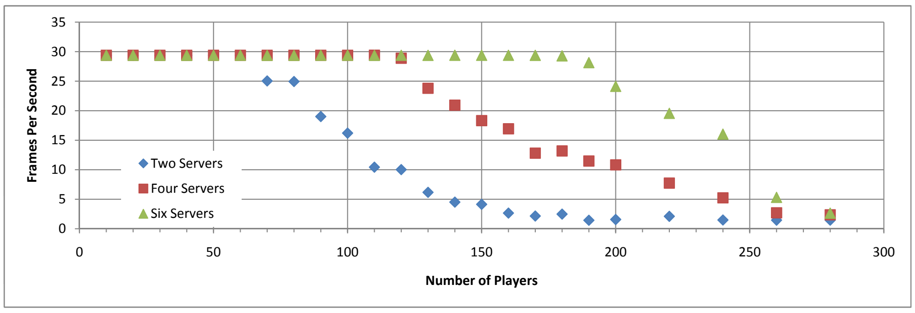


Figure 5.5: The combined frame rate graph for two, four, and six servers with the static solution in scenario A.

The reason that two servers fail at 80 players is because that one of the servers suddenly experiences an increase in the number of players and projectiles, and maintains this increase for a longer period of time. And as the players and projectiles increase, so does the required processing power. When the processing power of the servers have been exhausted, the frame rate will begin to drop for the servers. The drop in frame rate for 80 players can be seen in Figure 5.6, where one of the servers drops significantly in frame rate, three times.

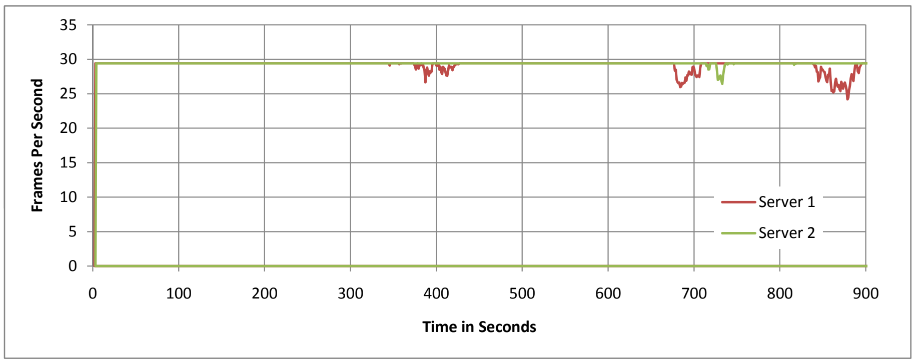
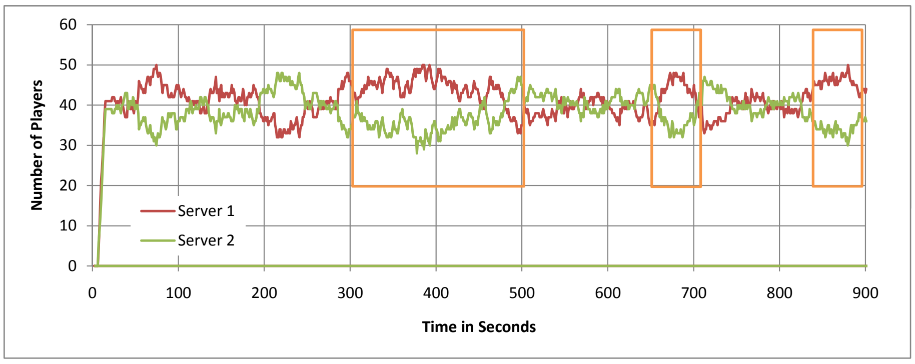


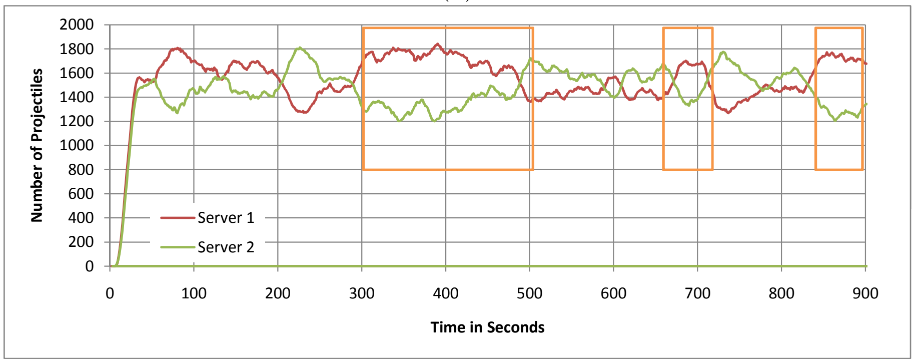
Figure 5.6: The frame rate from two servers with 80 players and the sudden drop at certain points in time for the static solution in scenario A.

The reason for this drop can be seen in Figure 5.7 (a) and (b), where the three times that server 1 drops is marked with orange boxes in (a) and (b). It can be seen that server 1 has both an increase in players and projectiles for a longer period of time. However the drops, which is in the last marker, makes the frame rate drop below 25 frames per second. The number (frequency) of such points, where the frame rate drops, increases as the number of players increase, and already starts at the 70 players mark, where it drops once. At 90 players, the drop are almost continues.

Latency and Bandwidth: The sent bandwidth graphs for six servers with 200 players can be seen in Figure 5.8 (a) and (b). The sent bandwidth, that is depicted is the vast



(a)



(b)

Figure 5.7: Graphs from two servers with 80 *players connected, (a) shows players and (b) shows projectiles for the static solution in scenario A.*

majority of the total bandwidth. These graphs shows that the bandwidth is not saturated for any of the six servers. The total bandwidth does not exceed 10MByte/sec and thereby does not exceed the 1Gb NIC in the test machines. This was also expected as the collision detection is already a bottleneck at 200 players.

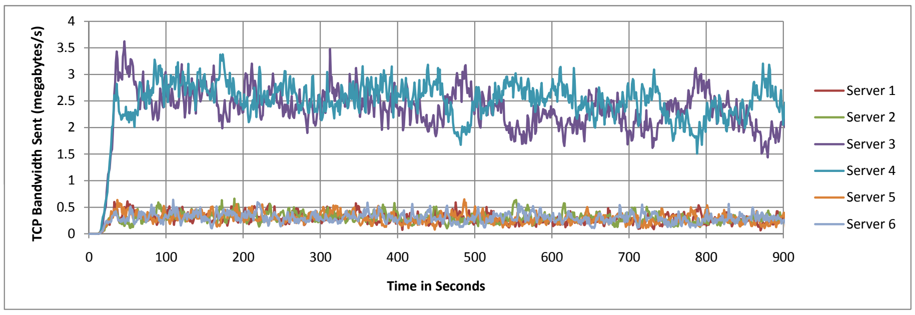
The bandwidth usage may seem very high. Note that clients will only receive a fraction of this bandwidth. This matters because players are often playing from their home and can therefore not be expected to have a very fast internet connection. It is however, important that the servers are connected to the internet through a very high bandwidth link. Also note that these tests are performed on a prototype and it should be possible to optimise sent bandwidth such that it is lower.

The latency does not exceed the limit of 100ms, which was the maximum amount of latency tolerable in FPS games, that had a latency profile of about $50 - 100$ ms. This can be seen in Figure 5.9, which shows a client from the six servers with 200 players test. Thus, it is again the CPU usage, which is the first bottleneck.

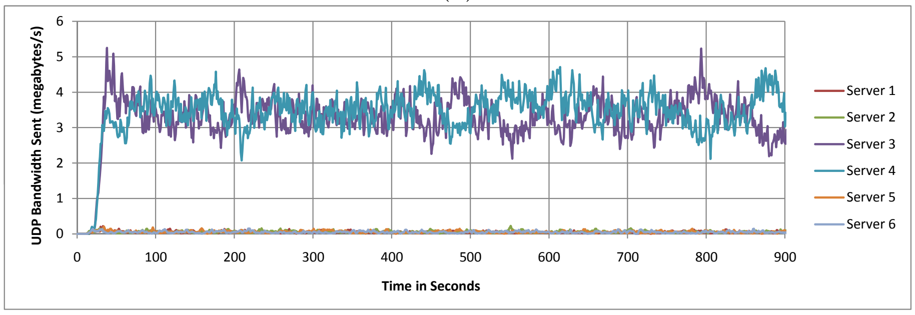
Comparison

The optimal solution, which used a single server to find the maximum number of players a single server could maintain, is compared against the results of the static multi-server solution. The results for optimal and static can be seen in Table 5.2.

The reason that the static multi-server solution did not obtain the same numbers as the optimal solution, is because of overhead to coordinate the overlapping regions. Because



(a)



(b)

Figure 5.8: Graphs with six servers and 200 *players for sent bandwidth for (a) TCP and (b) UDP for the static solution in scenario A.*

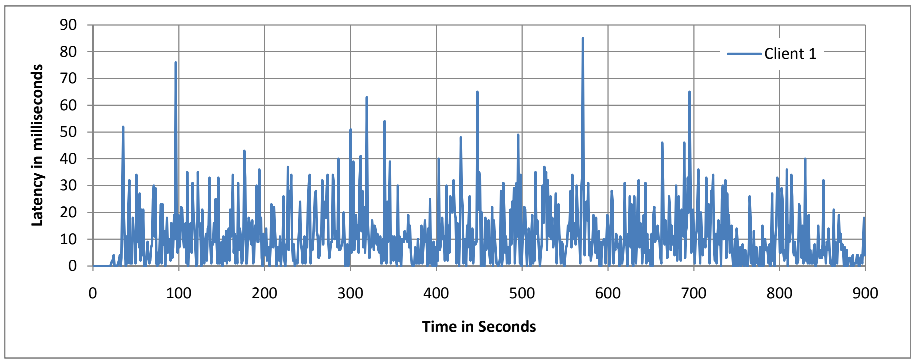


Figure 5.9: Latency for a single client in the test with six servers and 200 *players with the static solution in scenario A.*

Type	One server	Two servers	Four servers	Six servers
Optimal	48	96	192	288
Static		75*	127	196

*Table 5.2: The maximum number of players the optimal and static solution can handle in scenario A. * Average of the two border values.*

players movement is unpredictable, they do not create a perfect uniform distribution on the servers. This could be seen in Figure 5.7 (a) and (b), where one of the servers suddenly got an increase in both players and projectiles.

The static multi-server solution did show that an increase in the number of players can be achieved if the number of servers is increased. But the partitioning of the regions to the servers had to be good, such that the load could be distributed to all servers, and not create a server with more load than the others. The increase from two to four servers were 52 players, and from four to six servers, 69. These values are a bit lower than the expected 96 players increase, as achieved in the optimal solution.

5.4.3 Matrix

This section covers the extended Matrix solution tested in scenario A. It concerns the expected results of Matrix, and what the test actually showed. This is going to be evaluated to determine if Matrix performs as expected, and how it compares against the optimal and the static solutions. Matrix has been tested with two setups: Partitioning on the x-axis and partitioning on the y-axis. This is because "split-to-left" only works in one dimension at a time, and as this could have an affect on the results, both solutions are tested. Partitioning on the z-axis is not tested, because it is similar to that of the y-axis, because of the movement pattern in the test scenario.

Expected Results

We expect to have an increase in the number of players that can be handled, when we increase the number of servers. However, we do not expect it to perform better than the static solution, since the static solution was partitioned in such a way, that it should be able to handle a static load fairly well. The second reason is that the "split-to-left" technique only partitions in one dimension, which gives narrow and long regions, which might not be suitable as a partitioning of the virtual world.

We also expect the test to show that there is a little difference between the x-axis and the y-axis partitioning techniques. However x-axis could be marginally worse as it partitions perpendicular to the typical movement direction of the players in this test scenario. This could increase the number of times the players move between regions and worsen the performance.

Bandwidth and latency are expected to stay within the operational limits, as it is still the CPU usage which is the bottleneck.

Results

We have tested the extended Matrix solution from 10 to 280 players with an interval of 10 players with two, four, and six servers. The combined frame rate graph can be seen in Figure 5.10, where the "split-to-left" partitioning technique is on the x-axis.

It can be seen from the graph, that there is a big variance in the number of players supported by the number of servers. Two servers can handle 100 players, whereas four servers can only handle 120 players. This is because of how the "split-to-left" technique works and the setup of scenario A. When the first partition is done, then it equally distributes the workload on to two servers. However, when the second partition occurs, the two out-most pieces either contains very few or no players at all, which makes them severely underloaded. However, the two other servers that contains a high number of players are overloaded. The remaining underloaded servers has to merge with an overloaded server, which is not allowed.

From Figure 5.10, it can be seen that six servers can handle as many as 250 players, which is over twice the number of players, that four servers could handle. This sudden increase

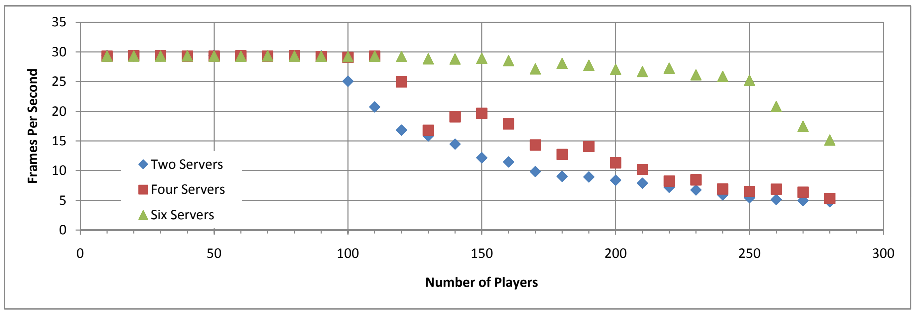
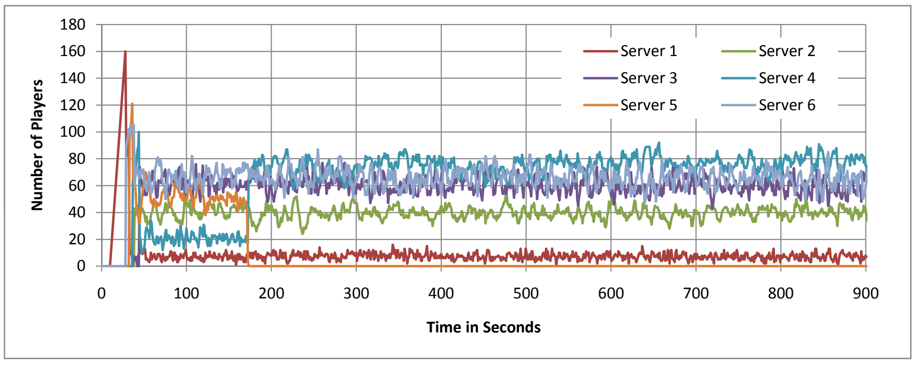
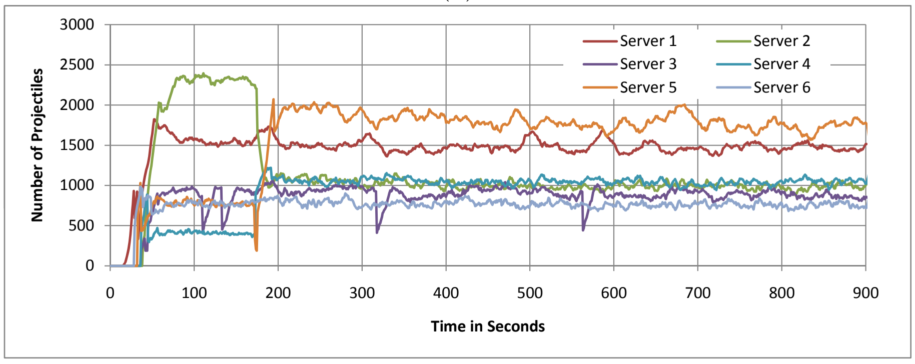


Figure 5.10: The combined frame rate graph for two, four, and six servers, where the matrix partitioning is on the x-axis. The graph is for test scenario A.

is because that the "split-to-left" technique can partition a third time, which enables it to distribute the workload evenly across four servers, and leaving the two outmost servers with few players, but many projectiles. This can be seen in Figure 5.11 (a) and (b), which contains two graphs for a run with six servers and 250 players. It can be seen in (a), that server 1 and server 5 contains either very few or none players, but in (b) they have a majority of the projectiles within the virtual world. These projectiles are the ones that were misses in the fight going on in the middle of the virtual world and therefore flies around at the outskirts, until their time limit has been reached. This is the reason that six servers can handle so many extra players than two and four servers.



(a)



(b)

Figure 5.11: Graphs from six servers with 250 players connected, (a) shows players and (b) shows projectiles using Matrix with x-axis partitioning in scenario A.

When partitioning the virtual world on another dimension, y-axis, the results are a bit different, as illustrated in Figure 5.12. For two and four servers the maximum player limit is approximately the same, where two servers can handle an extra 6 players. However, six servers can handle as many as 192 players, which is 58 players less than when partitioning on the x-axis. This is because only a few projectiles are moving to the two outmost regions, because of the moving direction of the players, whereas with the x-axis partition there would be a large amount of projectiles in the outmost regions. The choice of partitioning in either the x- or the y-axis is therefore important.

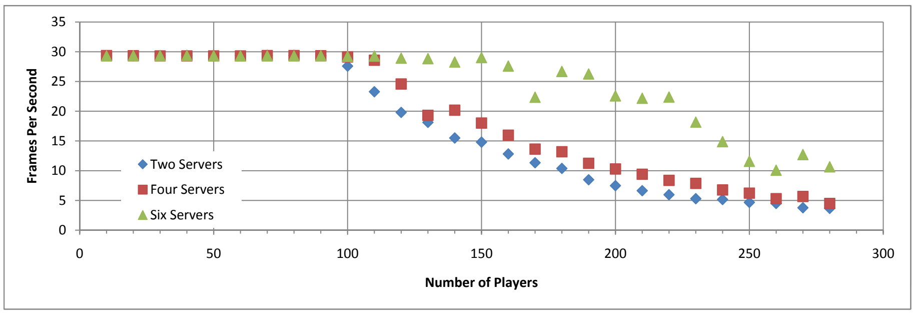


Figure 5.12: The combined frame rate graph for two, four, and six servers using Matrix with y-axis partitioning in scenario A.

As the latency and bandwidth are within operational limits for the tests where the frame rate is above 25, these graphs has been left out.

Comparison

Matrix handles around 100 players in both cases: x- and y-axis partitioning, which is about the same amount as the optimal. We recognise that Matrix can handle a lot of players with two servers, however with four servers we found that it can only handle 120 players, which is a very low increase in the number of players supported compared with the increase in number of servers. It is comparable with the static partitioning technique with four servers. The optimal solution suggested that four servers should be able to handle 192 players, which is not met. A surprise at six servers was noticed, where the x-axis partitioning could handle nearly an optimal number of players. Table 5.3 shows a comparison between the different methods tested, so far. It can be seen that the Matrix solution performs approximately the same as the static solution.

Type	One server	Two servers	Four servers	Six servers
Optimal	48	96	192	288
Static		75	127	196
Matrix x		100	120	250
Matrix y		106	119	192

Table 5.3: The maximum number of players the optimal, static, and Matrix solutions could handle in scenario A.

The huge increase for six servers was because two servers had a very large amount of the projectiles in the virtual world, but had no players. As mentioned, collision detection is only performed between players and projectiles. If, e.g. the number of projectiles is very low on a specific server, the huge amount of players that might be on the server does not contribute significantly towards the load. Similarly, if the number of players is very low, a high number of projectiles does not contribute significantly.

As this result shows rare actual real life scenario, we have chosen to run a series of tests with six servers with full collision detection. This means every object counts equally towards load on the server. This full collision detection enables projectiles to collide with other projectiles and spaceships to collide with other spaceships. We believe it to be a realistic test, as we assume that in a full size virtual world there would be other objects in these regions, which have to be tested for collision with the projectiles. The results from the full collision detection tests are described in Section 5.5.

The increase in players supported from two to four servers with Matrix on the $x-$ and y-axes are both around $13 - 20$ players. The increase from four to six servers is for the y-axis, 73 players, however the increase for the $x - axis$ was a whole 130 players, which is abnormal. This abnormality was previously explained.

5.4.4 JoHNUM

In this section the test of the JoHNUM techniques in scenario A will be evaluated. This section follows the same structure as the previous test sections.

Expected Results

We expect JoHNUM to be able to handle more players than Matrix. The reason for the expected better results for JoHNUM against Matrix is based on the more complex partitioning scheme. JoHNUM uses the population to determine the best possible partitioning whereas Matrix uses a simple “split-to-left” technique and do not take into account the population.

Bandwidth and latency are expected to stay within the operational limits, as it is still the CPU usage which is the bottleneck.

Results

We have tested JoHNUM from 10 to 280 players with two, four, and six servers. The combined frame rate graph for the servers can be seen in Figure 5.13.

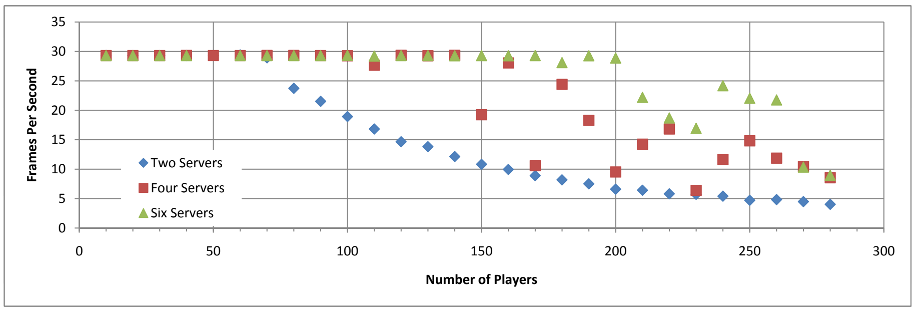


Figure 5.13: The combined frames per second graph for two, four, and six servers when using JoHNUM in scenario A.

It can be seen, that two servers can handle 78 players, which is 18 players below what the optimal solution suggested. However four servers could handle 143 players, but it can be seen that the 160 players mark is above the 25 frames per second limit, but this has been disregarded since 150 mark is below. This can be due to either erroneous data or because the algorithm took a bad decision, which was fatal for the results.

A bad decision is when the partitioning techniques creates a series of partitions and merges, and possibly creates a locked configuration, where one server is overload, whereas the others are either underloaded or averagely loaded, but can not merge to help the overloaded server. This problem is illustrated in Figure 5.14, which is the run with four servers and 170 players. After creating a few partitions, where all four servers are used to handle the load, it enters a state where it can not merge or partition to alleviate the load from the server, that has the majority of the load.

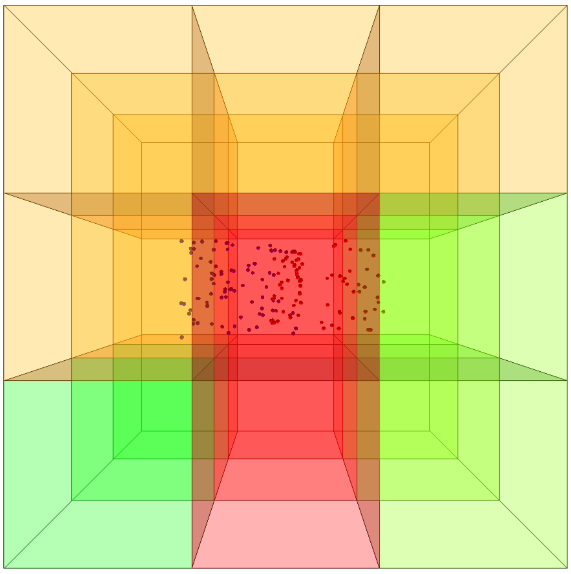


Figure 5.14: Test run with four servers and 170 *players, that has reached a configuration that can not merge or partition to help the overloaded server. This is with the JoHNUM solution in scenario A.*

The six servers can handle 205 players, which is below the optimal, but still has a support for an additional 62 players compared with four servers. Again it can be seen that 240 players is close to the 25 frames per second mark, whereas $210-230$ are lower. This can be due to a bad decision from the algorithm, which again creates a locked configuration. This locked configuration only happens if there are no more servers available to handle the load.

As the latency and bandwidth are within operational limits for the tests where frame rate is above 25 frames per second, these graphs have been left out.

Comparison

JoHNUM could handle 78 players with two servers, which is as many as the static solution, but not as many as Matrix. Matrix surprised with supporting more than the optimal with two servers, even though it created a partitioning that was equal to the static one. We still consider this a good result for JoHNUM. JoHNUM could support 143 players with four servers, which is better than both static and Matrix. With six servers it could handle as many as 205 players, which is better than the static, but not as good as Matrix and the optimal solution. At a certain point, JoHNUM had an increase in frame rate at 240 players, which was due to better choices when partitioning and merging regions. If better choices could be made, then JoHNUM might be able to support a bit more than the 205 players. A table of the different solutions and how they perform can be seen in Table 5.4.

Type	One server	Two servers	Four servers	Six servers
Optimal	48	96	192	288
Static		75	127	196
Matrix (x-axis)		100	120	250
Matrix (y-axis)		106	119	192
JoHNUM		78	143	205

Table 5.4: The maximum number of players the optimal, static, Matrix, and JoHNUM solutions could handle in scenario A.

5.5 Test Results: Scenario A - Full Collision Detection

This section covers the additional test identified in the previous section. As apparent from the previous section, Matrix received a sudden burst in number of supported players with six servers. This is due to the fact that most of the projectiles were in the two outermost regions, whereas all players were in the four inner regions. The game mechanics only performed collision detection between players and projectiles (players were free to fly through each other). This allowed us to support a larger number of players.

We have decided to test all techniques with full collision detection, where there is player/player, player/projectile, and projectile/projectile collision detection, so all objects now contribute to the load created from collision detection. This section details the results obtained.

5.5.1 Static Region Partitioning

The static region partitioning solution with full collision detection is illustrated in Figure 5.15. It is a combined graph with six servers running with 10 to 200 players with an interval of 10.

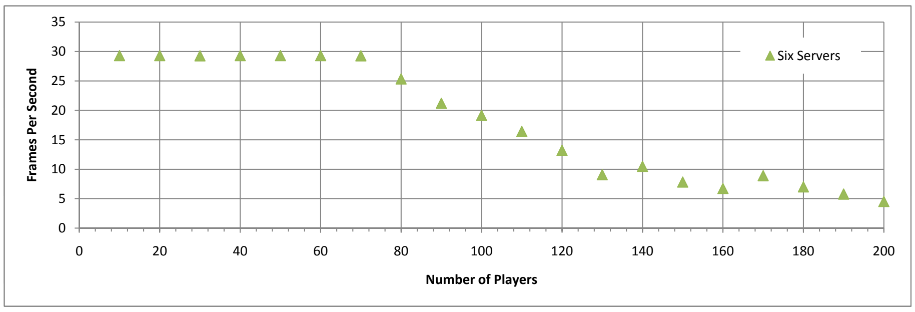


Figure 5.15: Combined full collision detection graph running static partitioning technique with six servers with 10 to 200 players in scenario A.

It can be seen that the number of players supported drops compared to the test without full collision detection. It can now support 80 players with six servers. This is a good result, because of the partitioning of the virtual world designed for this test case.

5.5.2 Matrix

Matrix has also been tested with full collision detection, both with x- and y-axis partitioning, as illustrated for x-axis in Figure 5.16 and y-axis in Figure 5.17.

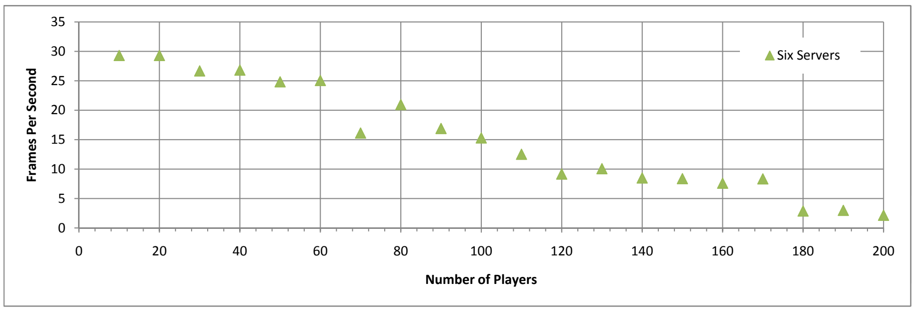


Figure 5.16: Combined full collision detection graph running Matrix x-axis partitioning with six servers with 10 *to* 200 *players in scenario A.*

In the previous test, Matrix performed better than the static partitioning with six servers in both types of partitions. However, in the x-axis partitioning illustrated in Figure 5.16, it performs worse than the static, where it supports about 55 players. This is even worse with the y-axis partitioning illustrated in Figure 5.17, where it only supports 35 players.

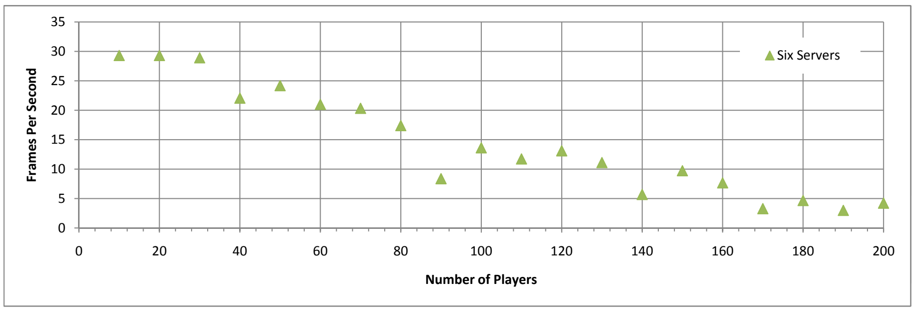


Figure 5.17: Combined full collision detection graph running Matrix y-axis partitioning with six servers with 10 *to* 200 *players in scenario A.*

5.5.3 JoHNUM

Lastly, JoHNUM has been tested with full collision detection with six servers from 10 to 200 players with a 10 player interval, as illustrated in Figure 5.18.

JoHNUM could in the previous test support approximately the same amount of players with six servers, as the static solution, which was $196 - 205$ players. However as illustrated in Figure 5.18 it can only support 26 players, which is much lower than both static and Matrix. This is due to oscillation in the decision making for partitioning and merging of regions, as illustrated in Figure 5.19 with CPU load versus time. The boxes in the top illustrates a time point, where a server either partitions or merges. It can be seen that from $50 - 100$ seconds it does 22 partitions and merges. Then from $100 - 450$ seconds it does 40 partitions and merges. At last, it gets to a stable partitioning of the work load, where it does not partition or merge.

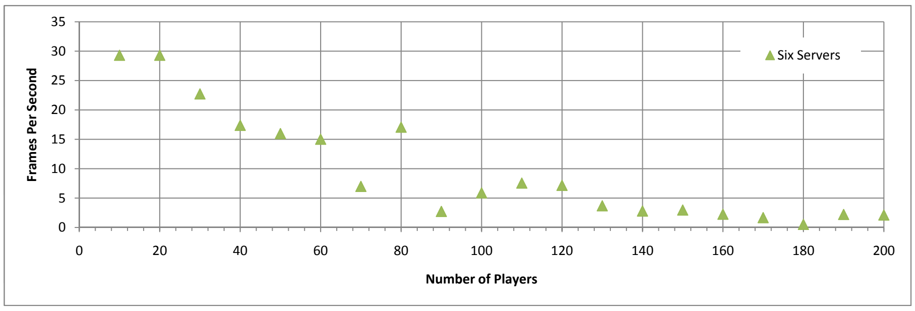


Figure 5.18: Combined full collision detection graph running JoHNUM with six servers with 10 *to* 200 *players in scenario A.*

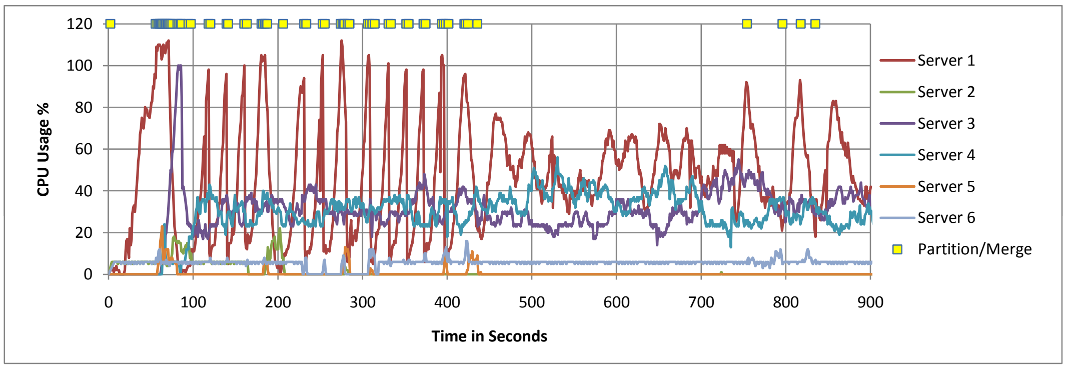


Figure 5.19: Oscillation seen from the test run with six servers and 30 *players with JoHNUM, where it does* 62 *partitions and merges within* 400 *seconds.*

The oscillations in the first 450 seconds in Figure 5.19 are due to the threshold for when a server can partition or merge. If these are not set appropriately, then oscillation will occur. We have tested another threshold value, which is illustrated in Figure 5.20, where no oscillation occurs, and thereby supports as many as 53 players. This is between the two partitionings from Matrix. However it is not as good as the static solution.

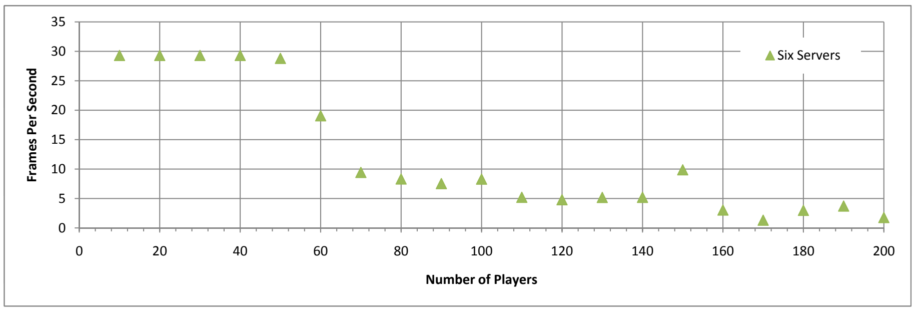


Figure 5.20: Combined full collision detection graph running JoHNUM in scenario A with six servers with 10 *to* 200 *players, but with no oscillations in the test runs.*

5.5.4 Comparison

In Table 5.5 it can be seen that the static solution performs best. This is because the virtual world is partitioned evenly between the servers around the area of the fight. This gives a good distribution for all the objects, as the servers in the middle handles the players, and the servers on the edges handles a large part of the projectiles.

Type	Six servers
Static	80
Matrix (x-axis)	55
Matrix (y-axis)	35
JoHNUM (oscillations)	26
JoHNUM	53

Table 5.5: Maximum number of players supported for the full collision detection test in scenario A.

Of the dynamic solutions, both JoHNUM and Matrix on the x-axis performed the best. However, on the y-axis performs worse, due to bad partitioning of the virtual world.

5.6 Test Results: Scenario B

This section presents the test results for test scenario B, described in Section 5.2. The results covers a single server solution, and the extended multi-server solutions: Static, Matrix, and JoHNUM, which will be compared against each other. Note that these tests are run with collision detection only between projectiles and players. This is not a full-collision detection test.

The single server solution is as in test scenario A only used to define the capacity of the optimal solution. Since the two test scenarios differs, we have chosen to test the single server separately from test scenario A. The main reason for doing a separate single server test in test scenario B, is because of the moving hotspot. The movement pattern of the avatars might give another result, than in test scenario A, where the movement pattern was head-on all the time.

5.6.1 Optimal

We tested the single server in test scenario B to determine its maximum capacity, before it drops below the 25 frames per second mark. It was tested from 10 to 280 players with an interval of 10, as illustrated in Figure 5.21.

The single server solution reached its maximum capacity at 131 players, which is a lot more than the single server solution in test scenario A. This is because of the moving hotspot, which reduces head-on collisions between the two teams. Instead it is in an angle, which concentrates the players within a small area, which yields higher accuracy from the players. This yields more hits, which in turn creates fewer projectiles. This angle is achieved because the spawning location is not rotated to face the new hotspot location, as it is in test scenario A. This therefore gives a more angled intersection of the two teams movement pattern. This is also the reason for the higher accuracy, since the two teams will fly longer and towards a focal point, thereby concentrating the two teams a bit more. If the 131 players for the

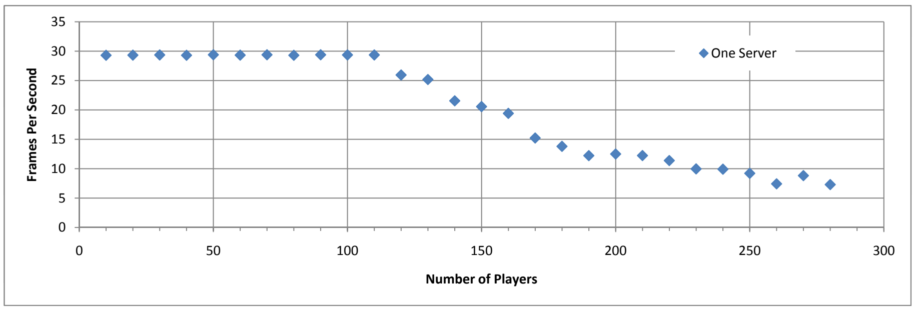


Figure 5.21: Illustration of a single server in scenario B, handling players from 10 *to* 280*, where the frame rate has dropped below* 25 *frames per second between* 130 *and* 140 *players.*

single server is scaled linearly, then this gives: 262 for two servers, 524 for four servers, and 786 for six servers, as described in Table 5.6.

Type	One server	Two servers	Four servers	Six servers
Optimal	131	262∗	524∗	786∗

*Table 5.6: The maximum number of players for one server, which has been linearly extrapolated to the optimal solution for two, four, and six servers for test scenario B. * Extrapolated data.*

5.6.2 Static Region Partitioning

This section treats the results obtained using static partitioning in test scenario B. Similar to Section 5.4.2 it will begin with the expected results, then present the actual results, and finally give a comparison of the results versus the expected results and the optimal solution.

Expected Results

With test scenario A, we expected the static partitioning to perform well, however in test scenario B this is not expected. The reason for this is because of the movement of the point of interest, that will create new hotspots in the virtual world. We expect the static partitioning to have approximately the same performance as a single server, because it can not react to moving hotspots. That means that we expect that the static partitioning will perform worse in test scenario B, than it did in test scenario A.

Bandwidth and latency are expected to stay within the operational limits, as it is still the CPU usage which is the bottleneck. We will check this in the log files to verify that this is also true for test scenario B.

Results

The static partitioning solution was tested with two, four, and six servers from 10 to 280 players with an interval of 10, like in test scenario A.

The combined results for two, four, and six servers is illustrated in Figure 5.22. It is apparent that the two servers setup drops below 25 frames per second at 132 players. Four servers drop below at 143 players, and six servers drop below at 176 players. The reason for such a small increase in number of players supported against the number of added servers,

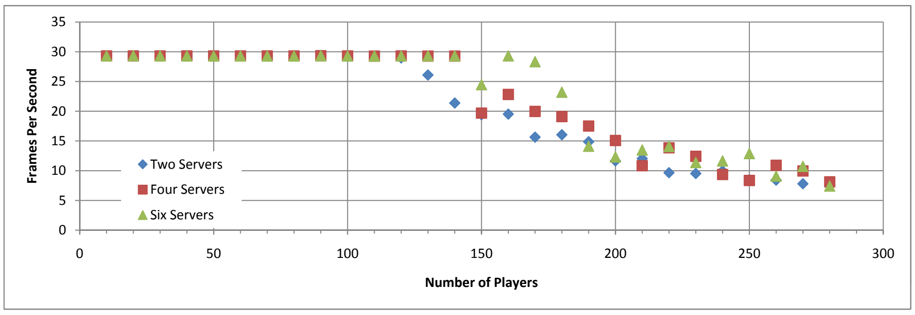


Figure 5.22: The combined frame rate graph for two, four, and six servers using static partitioning in scenario B.

is because the regions created are static and can not accommodate the movement of the hotspot. However, there is still a small increase, which is because of the movement pattern of the avatars. The avatars still spawn in the same area, as in test scenario A, so when they get killed and respawn, they spawn in another region, than the hotspot currently resides in. This distributes the load a bit, between several servers, as illustrated in Figure 5.23 (a-c). However it is still not enough to create a big difference when adding more servers.

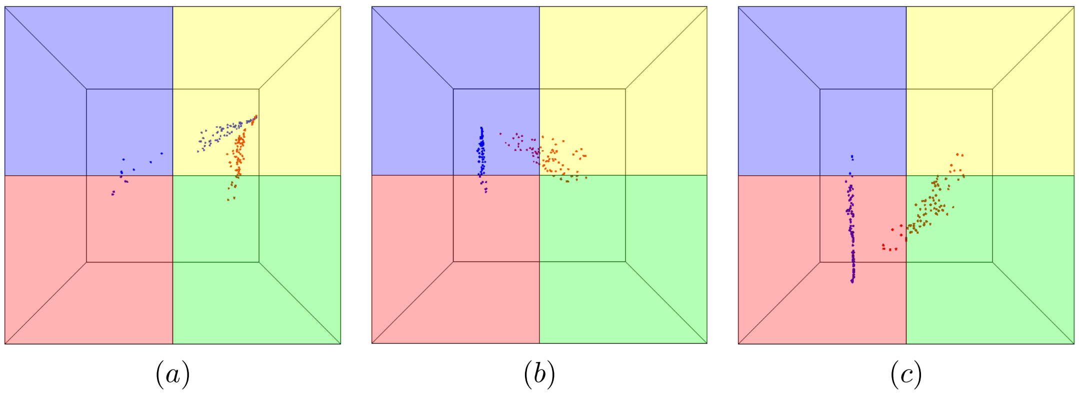


Figure 5.23: Moving the location of the hotspot gives three different movement patterns as illustrated in (a), (b), and (c).

As the latency and bandwidth are within operational limits for the tests results, where the frame rate is above 25 frames per second, we have omitted these graphs.

Comparison

The static partitioning solution gives a worse scalability, than was obtained in test scenario A. This is due to the moving hotspot, where avatars flock to one region of the virtual world. This increases the number of projectiles and avatars within the region. The load is not distributed, so we can only take advantage of the processing power of either one or very few servers.

The results for the increase in number of players against the additions of servers does not create a linear increase, as the optimal solution suggests. It is actually very low compared, which can be seen in Table 5.7, where the static solution only has a very limited increase in the number of players gained with two servers.

Type	One server	Two servers	Four servers	Six servers
Optimal	131	262	524	786
Static		132	143	176

Table 5.7: The optimal and static partitionings solutions population limits for one, two, four, and six servers in test scenario B.

5.6.3 Matrix

This section details our obtained test results, when testing the extended Matrix solution with test scenario B. Our expected results will be presented, followed by the actual results obtained. Finally, a comparison of the results with the static partitioning and the optimal solution will be given.

Expected Results

When testing scenario A, we expected it to handle reasonable well. The same case applies here. Matrix has the benefits of being able to dynamically partition the virtual world to alleviate heavy load in certain areas. We however still expect that Matrix will have difficulties to distribute the load evenly, because it does not take into account where the avatars are positioned and can only partition in one dimension at a time.

Bandwidth and latency are expected to be within operational limits, as it is still the CPU usage which is the bottleneck.

Results

The extended Matrix design was tested with a x- and y-axis partitioning with two, four, and six servers from 10 to 280 players with an interval of 10 and then from 300 to 500 with an interval of 20.

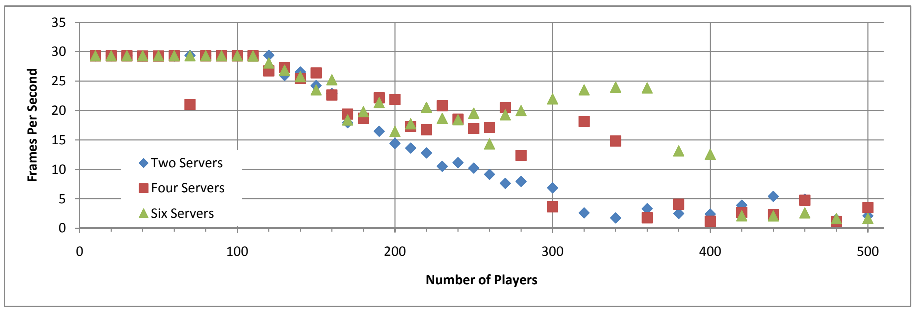


Figure 5.24: The combined frame rate graph for two, four, and six servers, where the partitioning is on the x-axis using Matrix in scenario B.

The combined results for the x-axis partitioning is illustrated in Figure 5.24 with two, four, and six servers. It can be seen in the graph, that two servers drops below a frame rate of 25 frames per second, at 145 players, four servers at 153 players, and six servers at 141 players.

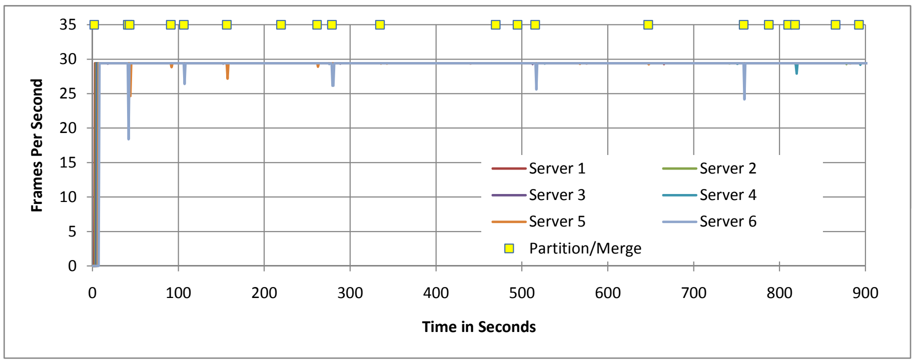


Figure 5.25: This graph shows frame rate as function of the running time of the experiment: Matrix solution with partitioning on the x-axis, 170 players and six servers in test scenario B.

The reason for the bad results with Matrix can be found by examining the graph in Figure 5.25 for six servers and 170 players, where a number of drops in frame rate is observed. These drops in frame rate occurs at times when the virtual world either partitions or merges. This is because Matrix's "split-to-left" technique gives the left-region to a new server, and all players or a majority of the players might be located in the left region, so it receives a large amount of information in a short time span, because of the migration of a large amount of players from one server to the new server. Such a large migration of players can be seen in Figure 5.26 (a-c), where the virtual world in (a) has partitioned into two regions and the majority of the players are in the left-region. But even when the virtual world has been partitioned, this might not alleviate the workload from one of the servers, so a second partition might be necessary, (b). However, by using the "split-to-left" technique a large amount of the players might be located in just one of the new regions, but as can be seen in (b) and (c) its about half of the players that are located within the left-region.

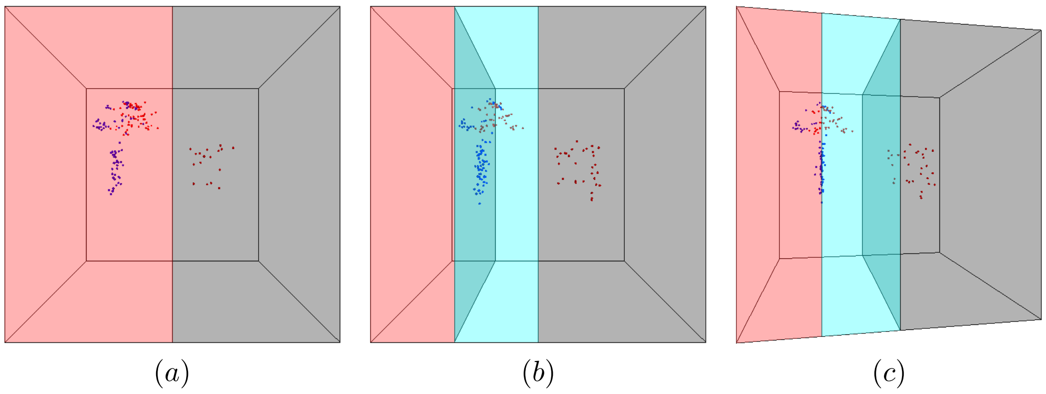


Figure 5.26: Illustration of a partitioning, where a sudden drop in frame rate has occurred, (a) is the initial state of the virtual world, (b) is the partitioning, where the left-region has the majority of the avatars, and (c) a rotated view of (b).

The drops in frame rate are not because of the server being overloaded, as can be seen in the graph of Figure 5.27. That is, the CPU usage is, when the drops occur, never greater than 80 – 90% CPU usage. However the drops are due to the transfer of a large amount of players to the new server, when partitioning. This can be seen in Figure 5.28, where the drops in frame rate are at the same time points, where there is a large transfer of players. This problem can be alleviated by only giving the new server partial controle of all the objects, and thereby use more time to give all the avatars to the new server. This was however not tested, so a moving average of the data points will be described in Section 5.7, where such single point drops are levelled out.

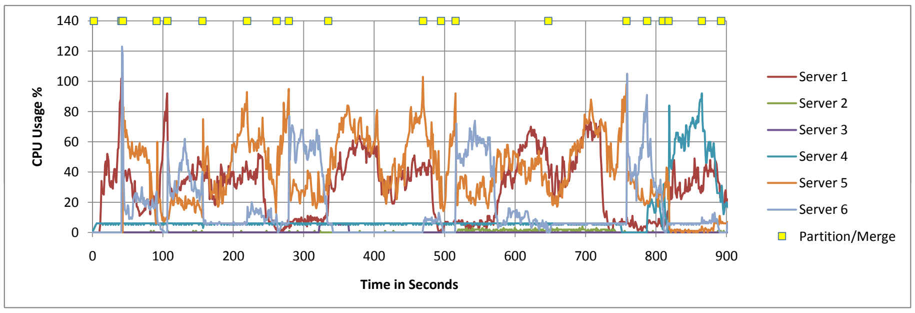


Figure 5.27: This graph shows CPU usage with relation to the running time of the experiment: Matrix solution with partitioning on the x-axis, 170 players, and six servers in test scenario B.

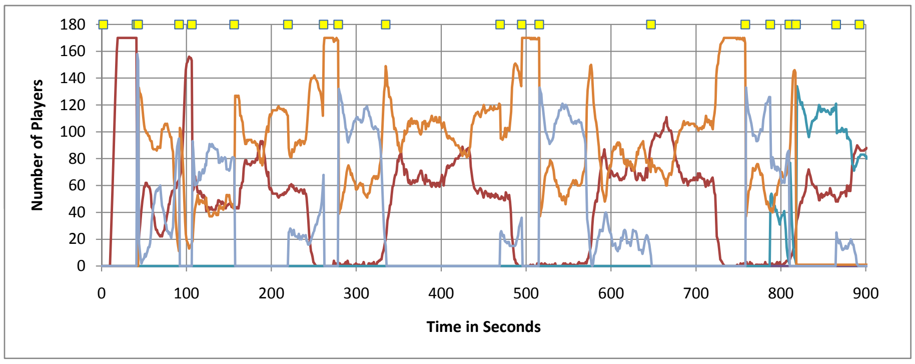


Figure 5.28: This graph shows the number of players with relation to the running time of the experiment: Matrix solution with partitioning on the x-axis, 170 players, and six servers in test scenario B.

The extended Matrix solution was likewise tested with a y-axis partitioning. The same problems as mentioned before is also seen in the y-axis partitioning, as illustrated in Figure 5.29. The frame rate drops below 25 frames per second, at 125 players for two servers, 126 players for four servers, and 115 players for six servers. There is not a big difference by adding more servers, but it has poorer performance than the x-axis partitioning.

Latency and bandwidth are within operational limits for the test results, where the frame rate is above 25 frames per second. Graphs for latency and bandwidth have therefore been omitted.

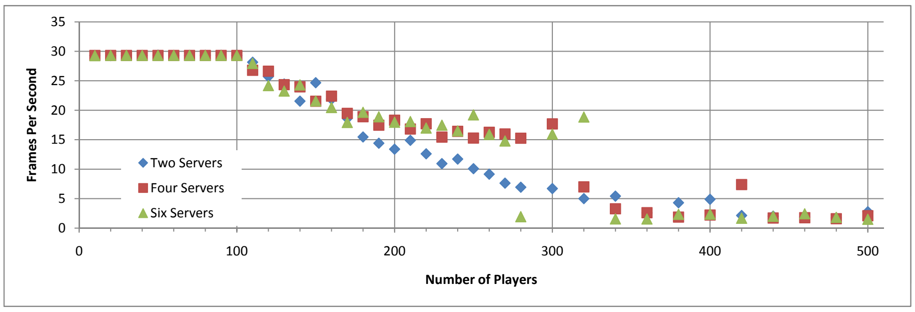


Figure 5.29: The combined frame rate graph for two, four, and six servers, where the partitioning is on the y-axis in test scenario B.

Comparison

The two Matrix partitionings showed degraded scalability results compared to the optimal solution. This was due to the problem of receiving a large number of objects, when a partition occurred. However, it can be seen from the results that, even though there existed the problem of transferring a large amount of objects for both partitionings, the x-axis partitioning still gave a better result, than the y-axis. However, the x-axis partitioning was only marginally better with two and four servers, and with six servers it was worse, as can be seen in Table 5.8.

Given the poor results that Matrix had, it has been decided to create a further test section, where moving average has been applied to level out the drops in frame rate. This will be described after JoHNUM results have been presented for this test scenario.

Type	One server	Two servers	Four servers	Six servers
Optimal	131	262	524	786
Static		132	143	176
Matrix (x-axis)		145	153	141
Matrix (y-axis)		125	126	115

Table 5.8: The optimal, static, and the two Matrix partitionings solutions population limit for one, two, four, and six servers in test scenario B.

5.6.4 JoHNUM

This section shows the results obtained with the evaluation of test scenario B using the extended JoHNUM solution. It explains the results we expect to see and then the actual results. Finally, the results are compared to the optimal, extended Matrix, and the static region partitioning solutions.

Expected Results

The extended JoHNUM techniques should be able to handle a group of moving players, because of its dynamic partitioning of the virtual world. It also has the benefits of trying to create the most even partitioning of the load to several servers, because it examines the population of the regions.

We also expect JoHNUMs techniques to have a better utilisation of resources. That is, JoHNUM should be able to assign more processing resources, where it is needed, but also remove resources from where it is no longer needed. When a group of players collectively move away from an area consisting of multiple regions, JoHNUM should be able to merge two regions which are now less densely populated. This releases servers which can be used in other places in the virtual world.

Finally, we expect the JoHNUM techniques to perform better than the Matrix solution. This is because the JoHNUM techniques takes into account the distribution of players inside a particular region when partitioning, whereas Matrix always partitions a region into two equally sized regions.

Bandwidth and latency are expected to be within operational limits, as it is still the CPU usage which is the bottleneck.

Results

The extended JoHNUM solution has been tested with two, four, and six servers with 10 to 280 players with an interval of 10, and 300 to 500 with an interval of 20 players.

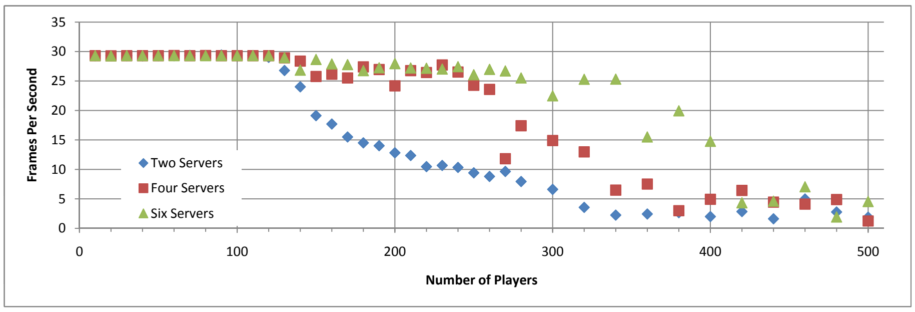


Figure 5.30: The combined frame rate graph for two, four, and six servers with JoHNUM and test scenario B.

The combined results for two, four, and six servers can be seen in Figure 5.30. It can be seen in the graph that JoHNUM supports 135 players with two servers, 247 players with four servers, and 281 with six servers. With six servers, there is a drop below 25 in frame rate at 300 players, however as can be seen in Figure 5.31, this drop happens at two points in the test run. Both drops are momentarily, and they happen just before 600 seconds and between 700 and 800 seconds, respectively. This may be because of the same reason as what happened in Matrix, where a large number of players needs to be transferred to a new server, because of a partitioning. This problem occurred at the $120 - 150$ player range in Matrix, and as this happens at around 300 players in JoHNUM. It is possible that such a large amount of players to be transferred at once, also gives problems for JoHNUM. Therefore it is, as with Matrix, necessary to level out the drops, as will be described later in Section 5.7.

As the latency and bandwidth are within operational limits for the test results, where the frame rate is above 25 frames per second. Graphs for latency and bandwidth have therefore been omitted.

Comparison

The extended JoHNUM solution could support more players, each time a pair of servers was added to the server pool. However, it did not get close to support the amount of players, that

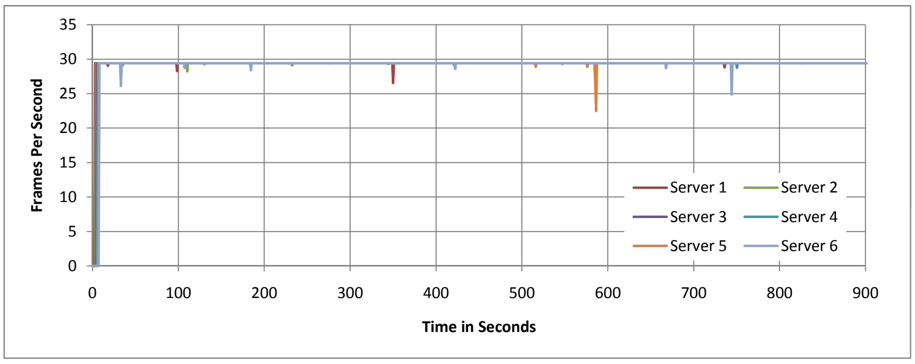


Figure 5.31: Graph showing a few drops in frame rate, due to a large amount of avatars being transferred to another server. This is for six servers and 300 players with JoHNUM, test scenario B.

the optimal solution suggested that might be possible, as can be seen in Table 5.9. It can be seen that with four and six servers, it actually supported more players than any of the other partitioning solutions. However, when the amount of players rose to about 300 and upwards, it had the same problem as was discovered with Matrix, which was momentarily drops in frame rate when transferring control of a large amount of players. A second evaluation of the tests will therefore be examined in the next section, where moving average is applied to the graphs.

Type	One server	Two servers	Four servers	Six servers
Optimal	131	262	524	786
Static		132	143	176
Matrix (x-axis)		145	153	141
Matrix (y-axis)		125	126	115
JoHNUM		135	247	281

Table 5.9: The optimal, static, the two Matrix partitioning, and JoHNUM solutions population limits for one, two, four, and six servers in test scenario B.

5.7 Test Results: Scenario B - Moving Average

This section will cover the test results from test scenario B, but with a moving average applied to frame rate measurements in order to give a more usable view of the graphs. The moving average is applied with a window size of three seconds. This is done to give a more accurate indication of how well the different partitioning techniques perform.

The argument against using moving average smoothening of the frame rate graphs is that sudden drops will not be apparent in the graphs. However, as the moving average is being applied with a window size of just three seconds, only very short-duration drops will be hidden from the graphs. These very short-duration drops are unlikely to substantially disrupt the experience for any players, due in part to the prediction performed on clients. Another approach could have been to just ignore measurements which are significantly lower than the next or previous measurement. However, this approach was not chosen due to the fact that serious drops in close succession could risk being ignored, even though players might notice continuous drops. Using the moving average, drops are still affecting the graph, and are not completely ignored. The three second window size was chosen because we estimate that only drops which are insignificant for the player experience will be hidden by the moving average when using the tree-second window size.

When determining how many players a technique can support with a given number of players, we look at the minimum frame rate for that experiment. If the minimum frame rate goes below 25 frames per second at any point, it is considered a failure to support that specific number of players. Thus, it makes sense to apply a moving average to level out the graphs, as long as the players do not notice the very short-duration drops that are being hidden. In a sense, we use the moving average to remove noise.

5.7.1 Static Multi-Server

The results for the static partitioning solution with applied moving average can be seen in Figure 5.32 for two, four, and six servers, which are from 10 to 280 players with an interval of 10.

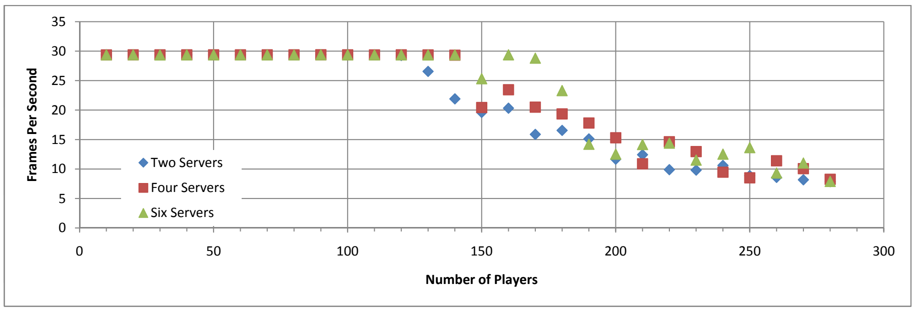


Figure 5.32: The frame rate graph for two, four, and six servers using the static partitioning solution in test scenario B applied moving average.

It can be seen from the graph in Figure 5.32, that the moving average has not increased the number of players that the different amount of servers can support. With two server we can still support around 133 players, likewise four servers around 144, and six servers around 178. This is because the static partitioning solution does not partition or merge the virtual world during runtime, so it will not be affected by a huge amount of transfers of objects to a new server.

5.7.2 Matrix

The extended Matrix solution was tested with x- and y-axis partitioning with applied moving average to the results, which can be seen in Figure 5.33 and Figure 5.34.

The results with x-axis partitioning, which can be seen in Figure 5.33, can now support 149 with two servers and $200 - 230$ players with four servers. The reason that four servers are set to have an interval of $200 - 230$ is because they are all just on the border of failing. Finally we can support 360 players with six servers. We disregard the point single point at 260 players because it is followed by five points above 25 frame per second. The reason that it fluctuates for all three server configurations is that Matrix might yield a poor partitioning, which can affect the results.

The results with y-axis partitioning can be seen in Figure 5.34, where two servers can support 153 players, four servers can support 300 players, and finally six servers can support

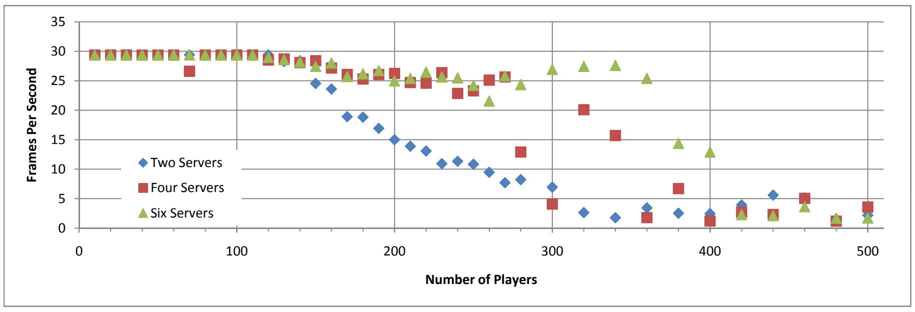


Figure 5.33: The frame rate graph for two, four, and six servers using the Matrix solution with x-axis partitioning in test scenario B applied moving average.

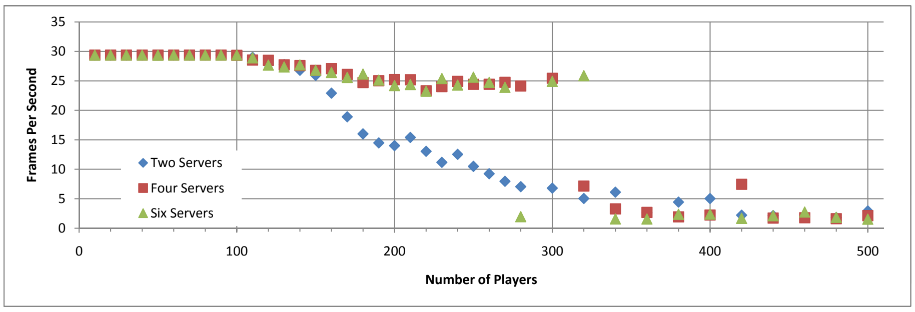


Figure 5.34: The frame rate graph for two, four, and six servers using the Matrix solution with y-axis partitioning in test scenario B applied moving average.

320, where 280 players is an erroneous measurement. The reason for the huge ranges is because many of the values lie on the 25 frames per second border. However, given that certain higher populations is above the 25 frames per second marker, it has been decided that they are valid.

5.7.3 JoHNUM

The results for the extended JoHNUM solution is illustrated in Figure 5.35 with applied moving average.

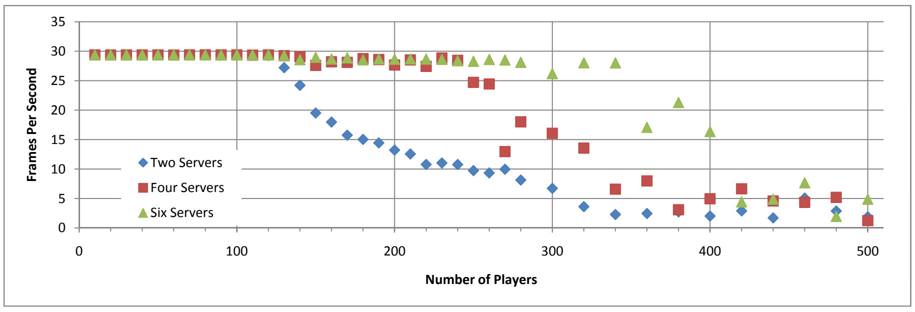


Figure 5.35: The frame rate graph for two, four, and six servers using the JoHNUM solution in test scenario B with applied moving average.

Two servers now supports 138 players, four servers supports 255 players, and six servers supports 348 players.

5.7.4 Comparison

The different partitioning solutions have been tested with test scenario B, where we applied a moving average on the data. The results can be seen in Table 5.10, where static and JoHNUM does not seem to have had a greater improvement. Matrix which had problems with transferring large amounts of players to a new server when partitioning. It can be seen that moving average had a big difference in the results for the Matrix solution. Both x- and y-axis partitioning in Matrix are better than the static partitioning solution, and they are either better or equally as good as JoHNUM. However, the ranges between the minimum and maximum supported players for two, four, and six servers varies much more, than against JoHNUM. This is because of the locked axis, when partitioning the virtual world.

Type	Two servers	Four servers	Six servers
Static	133	144	178
Matrix (x-axis)	149	200 – 230	360
Matrix (y-axis)	153	300	320
JoHNUM	138	255	348

Table 5.10: The static, the two Matrix partitioning, and JoHNUM approaches population limits for two, four, and six servers in test scenario B.

5.8 Summary

In this chapter we have described two test scenarios, a head-on and a hotspot scenario. The test environment include the hardware on which these tests were conducted has also been described. A general description of the two test scenarios and how they performed were given.

Test Case A: Full Collision	
Type	Six Servers
Static	80
Matrix (x-axis)	55
Matrix (y-axis)	35
JoHNUM (oscillation)	20
JoHNUM	53

Table 5.11: The maximum number of players supported with static, Matrix, and JoHNUM (with/without oscillation) for test scenario A with full collision detection.

The speedup in test scenario A from two servers to six servers is shown in Table 5.12. The speedup is calculated by taking the number of players from six servers and divided with the amount of players supported with two servers. This will give how many more players it can support by having three times more servers. From this table it can be seen that

JoHNUM has the best speedup, and if more servers would be added it should at some point be able to handle more players than the other solutions.

Type	Speedup by adding three times more servers
Static	196/75 = 2.61
Matrix (x-axis)	250/100 = 2.5
Matrix (y-axis)	192/106 = 1.81
JoHNUM	205/78 = 2.63

Table 5.12: Speedup by adding three times more servers in scenario A.

The optimal solution gave an impression of the linear performance, that might be possible. However all three multi-server solutions did not perform as well as the optimal suggested. This is illustrated in Figure 5.36, where all multi-server solutions are below the optimal. They are however very close to each other, where Matrix receives a sudden burst in number of players supported with six servers, where JoHNUM have more of a linear scaling.

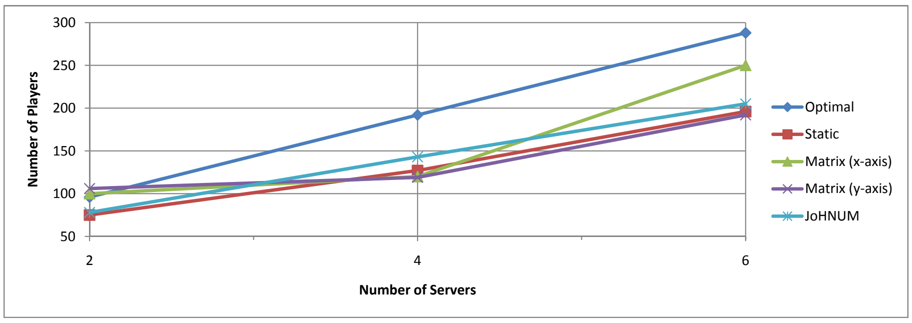


Figure 5.36: Relation between number of servers and how many players they can support with the different techniques in test scenario A.

In test scenario B, the static partitioning technique did not have a great impact, when adding more servers. This was because all the avatars would move towards one location, which resides in one region. The dynamic solutions could scale, when adding more servers, as can be seen from Table 5.13, where both Matrix and JoHNUM show greater scalability than the static technique. However, when partitioning the virtual world the results was affected by the large amount of transfer control information generated, when avatars were moved to a new server. This problem made Matrix perform worse than the static solution, but using moving average on the drops in frame rate showed that these drops were insignificant to players. JoHNUM could scale with several servers and performed better than both the static partitioning and Matrix. This was because JoHNUM was not affected as greatly by the amount of information that needed transfer when partitioning, because it tried to create as even a balance as possible by looking at the location of the avatars. JoHNUM is therefore well suited to handle moving hotspots within the virtual world, however it can not handle as many as the optimal solution.

To determine which technique is most suitable when creating a game depends on how the workload behaves. If it is known in advance how the game objects are distributed, the static technique is highly suitable, as could be seen in Figure 5.36. Another reason for the static technique to be preferred is the low level of complexity it takes to implement. In the

Type	Speedup by adding three times more servers
Static	178/133 = 1.34
Matrix (x-axis)	360/149 = 2.42
Matrix (y-axis)	320/153 = 2.09
JoHNUM	348/138 = 2.52

Table 5.13: Speedup by adding three times more servers in scenario B.

dynamic techniques there have to be the ability to change and reassign regions on runtime, which gives issues like oscillation and spikes when transferring players as was seen in test scenario B. However if the workload moves around, as in scenario B, the static technique has major issues, as can be seen in Figure 5.37. Here the dynamic techniques, JoHNUM and Matrix are the best choices. The JoHNUM technique is marginally better than Matrix, but it has a higher level of complexity, however JoHNUM is still our recommendation. This is due to the problems like oscillation occurs in both techniques and are the major contributor to complexity. When this hurdle is overcome the slightly more complex JoHNUM technique should not be a major issue to implement. We however also suspect the Matrix technique to have issues as it partitions in slices, which could give potential problems in huge virtual worlds.

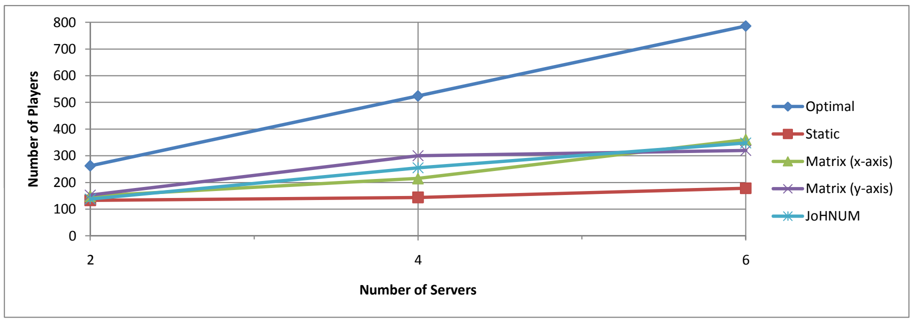


Figure 5.37: Relation between number of servers and how many players they can support with the different techniques in test scenario B.

Chapter 6

Conclusion

In this report we have looked upon the problem of scalability in MMOGs. We have done this by focusing on load distribution techniques, namely the static-, Matrix- and JoHNUM-techniques. We have also used a theoretical optimal technique to compare the performance against. Furthermore we have extended the Matrix and JoHNUM techniques to be applicable in a three-dimensional environment.

Subsequently, we developed a prototype game called Rock Pounder. The prototype served as a testbed for testing of the different partitioning techniques. The prototype was developed such that the underlying game engine was separate from the game specific implementation.

We performed tests with our extended editions of the Matrix and JoHNUM techniques, as Rock Pounder is a three-dimensional game. We conclude that our representations are true to the original techniques even though they have been extended to three dimensions. The tests were performed with two test scenarios, the first had the workload in a single area, and the second had the workload moving around between different areas. These tests were used to compare the different techniques against each other.

From the results we conclude that the static partitioning technique handles a workload in a single area well, both when there was full collision detection and when there was partial collision detection. This was due to the fact that the static partitioning we chose fit the test scenario well. The static technique is also more simple, compared to the dynamic techniques, because partitioning cannot be performed at runtime. Therefore, we recommend the static partitioning technique when the areas in which objects reside in is known. This makes is possible to define a good virtual world partitions. We found that static partitioning is not suited for dynamic virtual worlds, or for user created virtual worlds, as it will be hard to predict movement patterns in these situations.

We also conclude that the static solution is poor when the workload moves, and instead we recommend a dynamic solution. Of the dynamic solutions, the JoHNUM solution was only marginally better than the Matrix solution in our tests, and JoHNUM is more complex to implement than Matrix. Regardless, we still recommend JoHNUM as we suspect Matrix to have serious issues with large virtual worlds due to the fact that Matrix will produce slices which will span the whole width of the virtual world. Also, Matrix does not take into account the distribution of objects within the region when partitioning. However, this needs further investigation.

Lastly, we discovered some interesting facts from the tests, which should be taken into account when creating a dynamic load distribution technique for a game.

The first issue is apparent when a partition or merge is performed. We found that it was very important that objects which are to be moved from one server to another, are moved in a manner which does not produce a significant increase in workload on the servers. This has to be done to avoid that players have their game experience obstructed. This issue should be investigated further to find a good solution.

Another issue is to avoid oscillating partitioning and merging. We experienced this issue when the thresholds for when to partition and merge were inappropriately set. It is important to avoid oscillations as the servers involved will spend resources excessively. We solved the problem by choosing some more restrictive threshold values for partitioning and merging. To find a general method whereas oscillations can be avoided requires further investigation into the subject.

We found that it is possible to achieve fairly good scalability in MMOGs by using the load distribution techniques presented in this report.

6.1 Future Work

During the course of this project we identified several interesting paths that we would like to investigate in the future.

Smooth Object Transfer

We found that there could arise load spikes when a partition occurred as many objects were transferred from one server to another in a short time period. A method where objects are smoothly handed over from one server to another is an interesting point of further study. One possible idea to investigate could be to simply set a limit for the number of objects which can be can be transferred for a specific time period. This could help even out load spikes due to object transfers.

How to Avoid Oscillations

We also found that oscillations between partitioning and merging could be devastating for performance. Therefore it is interesting to investigate further how oscillations can be detected and avoided. This investigation could be done in a generic fashion to be able to utilise it for more than games, but for all kinds of dynamic load distribution systems. A possible solution worth investigating is automatic tuning of thresholds. If many partitions/merges are detected, thresholds could be automatically changed to avoid oscillations.

Human Players

It could also be interesting to attempt to evaluate partitioning schemes with purely human players to get completely realistic interactions and movements in the virtual world. Using real human players instead of our test clients, would make for interesting evaluation data in addition to the synthetic movement pattern we have already evaluated. However, as movement patterns are very game specific, the results might be hard to generalise.

Larger Scale

Our evaluations were conducted using up to 35 machines of which only 6 were used as servers. It could be interesting to apply our prototype implementations on a much larger number of machines, in order to have more servers and clients in the experiments at any one time. This will enable us to better evaluate how the techniques behaves when scaled up even further than what is presented here.

Graph Based Partitioning

It is possible to model the partitioning problem as a graph problem. Vertices corresponds to regions of the virtual world. Vertices have edges to vertices which represent adjacent regions in the virtual world. Edges are weighted according to the rate of object exchanges happening between two regions, corresponding to the vertices connected by the edge, as illustrated in Figure 6.1. Each server can be assigned multiple regions.

The partitioning problem becomes a problem of finding a partitioning of the graph such that region population is as balanced as possible, and object exchanges as minimised as possible. The importance of object movement patterns when partitioning virtual worlds of MMOGs has been repeated throughout this report. Further, it has been emphasised that movement patterns are game specific. The graph based partitioning technique tries to cater for this. Factors denoting the importance of minimising object exchanges (X) and the importance of balancing the number of objects (B) across partitions are applied when determining which partitioning of the graphs is preferred. Factors X and B can be considered *knobs* for tuning the partitioning algorithm to fit the characteristics of a specific implementation. E.g. if it is determined that object exchanges are very disturbing to the player experience, factor X corresponding to the importance of minimising object exchanges could be increased, making object exchanges weigh heavier in the ordering of partitions.

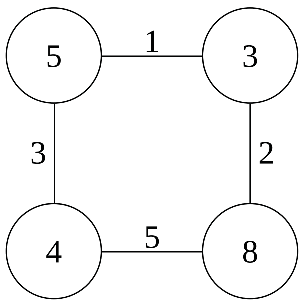


Figure 6.1: Illustration of a graph-based partitioning solution, where $m = 4$.

In practice the number of available servers n determine how many partitions the graph should be partitioned into. Each partition $P_1, P_2, ..., P_n$ is a set of regions. Regions are denoted $R_1, R_2, ..., R_m$ for a virtual world with m regions. These m regions correspond to m vertices in the graph.

The number of regions m in the virtual world is also a parameter which can be tuned. More regions should give a more fine-grained treatment of the virtual world, but more regions will also mean that the problem of finding a good partition increases in the number of configurations which must be examined. Additionally, increasing the number of regions in the virtual world will increase the overhead incurred in game servers from having to keep track of an increased number of regions.

In other words, choice of the number of regions can be considered a trade-off between these two:

- Computing time it takes to find a good solution and the overhead incurred in game servers by increasing number of regions.
- Quality of the found solution

This model explicitly addresses the problem of object exchanges. Intensive object exchanges were shown to significantly increase the workload of servers in this report, and in general, they are an inevitable source of overhead. However, since the importance of object exchanges is an adjustable factor, the model is applicable even in implementations were the problem of object exchanges has been partly mitigated.

Implementation of this technique could be done by way of a simple parameterised equation denoting a number of points for each solution. The solutions are then ordered by points, and the one with the most points is chosen as the new virtual world partition. A different approach could be to devise a graph traversal algorithm wich takes into account the same parameters.

The technique could be implemented as a dynamic technique, just like the JoHNUM and Matrix techniques. If a server becomes overloaded it could trigger this partitioning algorithm for all of regions in the game world, or for specific areas of the game world. It should also be noted that this technique allows for a much more holistic analysis of the game world as opposed to Matrix and JoHNUM which almost exclusively deals with sub-sets of the virtual world when partitioning or merging. The option of applying a holistic view should make it possible to make much better decisions for the virtual world as a whole. E.g. in JoHNUM, after a partition has been performed, merging is restricted to servers on the same level in the tree structure used. Applying the graph based partitioning technique with a holistic view, would allow partitions/merges to be made, regardless of any previous partitions/merges.

We believe the graph based partitioning technique outlined here could allow for better load distribution than the Matrix and JoHNUM techniques for the reasons mentioned above.

Bibliography

[1] CCP Games. **EVE Online**. http://www.eveonline.com/.

[2] IGN.com. **Magic World Online Interview - Part 1**. http://rpgvault.ign.com/articles/873/873793p1.html.

[3] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. **A Distributed Architecture for Online Multiplayer Games**. http://www.cs.cmu.edu/~ashu/papers/nsdi2006.pdf.

[4] Ralf Steinmetz and Klara Nachstedt. *Multimedia: Computing, Communications & Applications*. 1995.

[5] Blizzard Entertainment. **World of Warcraft**. http://www.worldofwarcraft.com/index.xml.

[6] Anders Ejlersen, Rasmus Kristensen, and Ron Cohen. **Distributed Game Engine for Massively Multiplayer Online Shooting Games**. DAT5 study report, 2010.

[7] Rajesh Krishna Balan, Maria Ebling, Paul Castro, and Archan Misra. **Matrix: Adaptive Middleware for Distributed Multiplayer Games**. *Middleware 2005*, 2005.

[8] Umar Farooq and John Glauert. **Managing Scalability and Load Distribution for Large Scale Virtual Worlds**. http://www.uea.ac.uk/polopoly_fs/1.133529!Farooq.pdf.

[9] Umar Farooq and John Glauert. **ARA: An Aggregate Region Assignment Algorithm for Resource Minimization and Load Distribution in Virtual Worlds**. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5272118&isnumber=5272056.

[10] Unity Technologies. **Unity3D**. http://www.unity3d.com/.

[11] Luke Hodorowicz. **Elements of a Game Engine**. http://www.flipcode.com/archives/Elements_Of_A_Game_Engine.shtml.

[12] Bryan Ford, Pyda Srisuresh, and Dan Kegel. **Peer-to-Peer Communication Across Network Address Translators**. *USENIX Annual Technical Conference, 2005*, 2005.

[13] R. Fielding, H. Frystyk, T. Berners-Lee, J. Gettys, and Jeffrey C. Mogul. **Hypertext Transfer Protocol - HTTP/1.1**. 1996.

[14] ID Software. Quake iii arena, 1999.
http://www.idsoftware.com/games/quake/quake3-arena/.

[15] Beverly Yang and Hector Garcia-Molina. Comparing hybrid peer-to-peer systems. *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001.

[16] Sugih Jamin. **Networking Multiplayer Games**, 2005.
http://ai.eecs.umich.edu/soar/Classes/494/talks/lecture-15.pdf.

[17] S. Webb, S. Soh, and W.A. Perth. **A Survey on Network Game Cheats and P2P Solutions**. *Australian Journal of Intelligent Information Processing Systems*, 9(4):34–43, 2008.

[18] Massively.com. **EVE Online's Server Model**, 2008. http://www.massively.com/2008/09/28/eve-evolved-eve-onlines-server-model/.

[19] Wikipedia. **Multitier Architecture**, 2010.
http://en.wikipedia.org/wiki/Multitier_architecture.

[20] CCP Games. **EVE Online Architecture**.
http://www.eveonline.com/devblog.asp?a=blog&bid=286.

[21] EVE Online Developers. **StacklessIO**, 2008.
http://www.eveonline.com/devblog.asp?a=blog&bid=584.

[22] James M. Van Verth and Lars M. Bishop. *Essential Mathematics For Games & Interactive Applications*. Morgan Kaufmann Publishers, 2004.

[23] Jesse Aronson. **Dead Reckoning: Latency Hiding for Networked Games**, 1997. http://www.gamasutra.com/view/feature/3230/dead_reckoning_latency_hiding_for_.php.

Appendix **A**

Terminology

This section covers some of the game terminology used in the report:

Virtual world: A computerised representation of the world or a fantasy world, e.g. EVE Online uses space to create a virtual world and World of Warcraft uses a fantasy world.

Avatar: A controllable character within a game. In Rock Pounder it is a small spaceship, that can shoot. It is a games visualisation of either you, or a character within the games' story.

NPC: NPC stands for Non-Playable Character, which resembles an avatar, but is controlled by the AI (Artificial Intelligence) in the game.

Gameplay: A combined experience of the game and mechanics of the game. It considers how the player moves, interacts, visual style, story, and so on, of the virtual world.

Multiplayer: Refers to a game, where the virtual world can have avatars from more than one player. Multiplayer games have evolved from two players using one computer, to several thousands with each their computer.

MMO: MMO stands for "Mass-Multiplayer Online" and often refers to a game with thousands of players in its virtual world. A MMO game is often called a MMOG, where "G" stands for "Game". A MMOG consists of either a single or several persistent virtual worlds, where players can control their avatars within.

Shard: A shard is a single persistent world, where players can roam in. Also known as a duplicate world. World of Warcraft uses several shards, whereas EVE Online uses a single shard.

Game genres: There exist several game genres within the gaming industry. This report concerned the fast-paced nature of FPS (First-Person Shooter) games, but where EVE Online and World of Warcraft both are RPGs (Role-Playing Games). A game genre can be combined with MMO, which describes the game as a mass-multiplayer online game, e.g. MMORPG. There exist several other game genres, e.g. RTS (Real-Time Strategy).

Game Engine: A game engine is a system of components, which can be used in a game. Its components can be: Graphics rendering, sound, input, network, physics, and so on. The game engine provides an interface for the game logic, which is the content of the game: Models, sound, behaviour, what should be applied physics, and so on. There exist several commercial game engines and tools, e.g. Unity3D[11].

Game Loop: In a game engine, there is an essential part, which is the game loop. The game loop uses all the components within the game engine, and runs the action that needs to be done in the game, such that it can draw a frame of what has actually happened in the virtual world on the display of the player. The game loop, usually checks for input from the user, actions to be done with the input, check other behaviour within the game, e.g NPC behaviour, then finds the current graphics to draw on the player screen.

Field of View: Field of view is the area which a player can see. This area is a triangle or pyramid shape where the apex is at the players location. The bottom of the triangle or pyramid is located in the direction which the players is facing. The sides of pyramid are marks the limits of what the player can see.

Appendix B

Correspondace with Umar Farooq

Several e-mails were sent and received from Umar Farooq, one of the authors of the JoHNUM articles. One of the emails was about some key aspects of JoHNUM, so this e-mail has been attached below. Emphasised sections are text sent from Umar Farooq to us.

> I would like to thank you for sending the two articles. I had actually already searched IEEE and ACM, but I must have done something wrong. I have read the two papers and it made several things more clear for me. However I have a few questions and concerns, that I hope you might answer and discuss with me. All the questions refer to the paper: "Managing Scalability and Load Distribution for Large Scale Virtual Worlds", which I also has attached as a PDF.
>
> *You welcome! Thanks for your interest, would love to answer and discuss the questions you mentioned, please keep it in mind this work is in initial phases and we are working on the rest of the issues.*
>
> - [p.3, column 1, line 7-10] You mention that Matrix yields a larger RMT level and that compromises consistency. I do however acknowledge that a RMT with a smaller number of levels is good, but fail at how it yields better consistency. If a VE is partitioned onto several servers, then how would the consistency be better if the RMT has fewer levels? The VE is still partitioned onto the same number of servers if the tree either has a high or low number of levels. Does it not?
>
> *Point 1- This work has two major parts, and so far we have investigated scalability and load distribution. We are currently working on a consistency concerns and expecting better results with reduced number of hops and messages. Further, the Matrix compromises on consistency due to its design goal, targeting game environments.*
>
> - [p.3, column 1, line 29-30] You also mention that when a server is underutilized, then the child servers return to the parent server. However does that not create isolated regions in the parent server? E.g. if a parent server has four child servers, then a child server is merged with the parent server with regions in the

bottom left corner, but the parent has regions from the top left corner.

Point 2- We have worked on the split and assignment strategies while the merging process is not yet presented, however it considers the same rules avoiding isolated regions.

- [p.3, column 2, line 2-5] What is permanent assignment used for? You mention it is used for single smaller regions.
- [p.3, column 2, line 10-] You mention that the region is split into several smaller regions and that the RSF value is incremented until none of the smaller regions has a player value greater than the MSC. However does that not limit the RSF value to two and three, since the player value probably will fall under the MSC quite fast? And could that not lead to a new partitioning fast after the first, since the server might need to activate a new partitioning?

Point 3 and 4- The terms provisional and permanent indicate the status of the region, where a region with status permanent is candidate for further split. This mechanism avoids increase in number of levels in the RMT (JoHNUM: JAS4 compared with JAS3).

- [p.4, column 2, line 15-] You mention that you use two roots for the partitioning algorithm. The top-left and top-right. Is that because you want all combinations given a matrix of n*n?

Point 5- Yup, any two consecutive roots cover the possible favourable choices, however selecting other roots might require modified strategies.

I would like to thank you beforehand, because of my curiosity.
You mention that you have a third paper in the press. I would really like if you could send it to me, when you can. I hope its "A Dynamic Load Distribution Algorithm for Virtual Worlds", since I am rather curious about the communication model and the tests of it.

I hope this help you understand, would love to share further improvements and work later. We have developed an abstract communication model, but we might be able to present detailed results, once we finish our actual implementation (under development).

I am also curious about the paper: "Organization of the Existing Open Source VE Development Frameworks and their Evaluation". Is it possible to get this paper?

I am really sorry that won't be able to share the fourth paper with you soon, but if you need to know and discuss about the best open source framework to work with, I would love to share my experiences with you.

Appendix C

DVD Content

The DVD contains the following parts:

Code: The following code projects are included for static, Matrix, and JoHNUM:

- **GUI Client:** The GUI client for Rock Pounder written in C++, which includes Visual Studio C++ and XCode project files.
- **Test Client:** The test client for Rock Pounder written in C++, which includes Visual Studio C++ and XCode project files.
- **Server:** The game server for Rock Pounder written in C++, which includes Visual Studio C++ and XCode project files.
- **Coordinator:** The coordinator written in C++, which includes Visual Studio C++ and XCode project files.

Binaries: The different binaries for static, Matrix, and JoHNUM:

- **Windows:** Rock Pounder: GUI Client, test client, server, and coordinator binaries.
- **OSX:** Rock Pounder: GUI client, test client, server, and coordinator.
- **Ubuntu:** Rock Pounder: Test client, server, and coordinator. No GUI client is provided.

Test Results: The test results are provided in two formats:

- **RAW:** All test runs' CSV files are provided from the clients, server, and coordinator.
- **MS Excel:** The Excel files from all test runs are provided with all servers, coordinator, and for ten clients.

Tools: The tools developed during the project period:

- **Replay Simulation:** A tool developed in Unity3D 2.6 with JavaScript .Net. The tool can give a replay of a test run by reading the .CSV files and .CFG file or splitlog. It gives a 3D-view of the virtual world, where players motion within the virtual world can be seen. It gives a visual representation of the servers workload by using colours, green for low load and red for high load. It also provides a frame rate graph of all the servers.

- **Excel Conversion:** A small tool developed in Visual Studio 2010 with C#. The tool converts .CSV files from the test runs into .XLSX files (Excel format), where three workbooks are created: Clients, servers, and coordinator for each test run. It automatically inserts the information from the .CSV files into the .XLSX files, and afterwards creates some predefined graphs of the data.
- **Find Combinations:** A small tool developed in XCode with C. The tool finds all the unique combinations when using JoHNUMs partitioning scheme and outputs them to a .TXT file.
- **View Combination:** A tool developed in Unity3D 2.6 with JavaScript .Net. The tool provides a 3D view of a $3 \times 3 \times 3$ partitioning from JoHNUM. It reads a .TXT file with the solutions that is in need of preview. Two files are provided: Solutions2D.TXT and Solutions3D.TXT. These two files provide all the unique combinations for JoHNUM in 2D and 3D, where the criteria for JoHNUM are satisfied.
- **JoHNUM3D Unique:** A small tool developed in Visual Studio 2010 with C#. The tool uses the different partitioning techniques and scheme from JoHNUM, where it has been developed for 3D application. It counts all the unique combinations the four algorithms yields and the total number of unique combinations.

Videos: Simulations from both test scenarios with all three techniques: static, Matrix and JoHNUM captured from the Replay Simulation tool.

Report: The report is provided in PDF-format.

Repository: The SVN repository for the project.