Latency Measurement, Modelling and Management for Interactive Remote Rendering

Richard Cloete

Supervisor: Prof. N.J. Holliman

School of Computing
Newcastle University

This dissertation is submitted for the degree of
Doctor of Philosophy

Abstract

Interactive Remote Rendering (IRR) systems enable computationally intensive rendering tasks to be offloaded to powerful remote servers, while permitting real-time user interaction. By streaming only images from the server to the client, these systems solve many issues, but can be adversely affected by Interaction Latency (IL). This thesis explores the use of keyboard-based user interaction prediction as a potential method for reducing IL. Specifically, the following questions are addressed: What is the effect of prediction on IL? How can we model and simulate latency? How can we measure IL when prediction is used? What is the optimal number of predictions ahead required to minimise latency? On which side of the network should prediction be performed? The literature describes a few cases of prediction being used in IRR systems but there exists a lack of knowledge pertaining to the development, integration and measurement of prediction into such systems. Initial investigation identified a lack of robust techniques for simulating and measuring latency in IRR systems, especially those employing prediction. A latency model is introduced, and a simulator is developed, demonstrating results comparable to the real-world. Latency simulation is shown to be accurate and is integrated into a "IRR simulator platform", permitting the exploration of the above research questions. As a result, two novel latency measurement techniques are presented. A prediction module is then developed and used in conjunction with the simulator platform. Results show that IL can be substantially reduced but predicting too far ahead negatively impacts IL, while less interaction history is found to result in lower mean IL. Finally, Client-Side Prediction was found to be more favourable for IL with respect to the amount of interaction history used, while Server-Side Prediction is shown to facilitate lower IL when predicting more than one step ahead. The results and tools presented in this thesis should prove useful for future exploration of PIRR systems.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Richard Cloete

Acknowledgements & Dedication

I'd like to thank Professor Nick Holliman for supervising my PhD. Nick's guidance and support has been invaluable. His help in shaping my research and writing has been immeasurable and I am deeply grateful for all the time he has given to me. I'd also like to thank Professor Paul Watson and Professor Darren Wilkinson for the incredible opportunity they provided me with. Paul's counsel and openness with me is truly appreciated. I am also immensely grateful to my wife, Dr. Evgeniya Shmeleva, to whom I dedicate this work and my life, for her patience with me over the past 4 years, for listening to me ramble on about ideas, for supporting me emotionally and for being there for me when I needed her most. Broadly, I'd like to thank the entire CDT team and students for the teaching, support and friendships and kindness shown to me over the last 4 years. Finally, I'd like to thank the EPSRC for funding my PhD for without their generosity, I would not be where I am today.
Thank you everyone!

Table of Contents

List of Figures

List of Tables

Acronyms

CFB Client Frame Buffer. 65, 66, 94, 95

CIB Client Interaction Buffer. 63–66

CL Client Latency. 18, 28, 32

CSP Client-Side Prediction. 117–126, 129, 130, 132–135, 138–140, 142, 143, 145, 147, 148

DL Display Latency. 28, 29, 49

IDL Input Device Latency. 28, 49

IL Interaction Latency. 1–5, 15, 17–20, 22–30, 32–38, 40–42, 44–50, 52, 55, 60, 61, 63, 65, 70–73, 75–82, 85–87, 89–91, 93, 96–98, 100, 101, 103, 104, 106, 107, 109, 111–115, 117, 118, 120–126, 128–149

IRR Interactive Remote Rendering. 1–5, 8, 9, 14, 16, 17, 21, 22, 25–28, 30, 32–36, 38, 40–42, 44, 47–51, 55, 60, 61, 71, 78, 85, 91, 94, 95, 106–108, 114, 141, 142, 146–148

LMT Latency Measurement Tool. 27, 35, 36, 38–40, 44, 45, 47–50, 142

LOD Levels of Detail. 10

MPA Maximum Predictions Ahead. 69–72, 76–78, 80–83, 85, 86, 89–91, 93, 101, 103, 104, 106, 109–115, 119, 120, 125, 129, 130, 132–140, 143–145, 148

NL Network Latency. 3, 18, 22–24, 28, 32, 34, 40, 68, 69, 72–74, 76, 77, 82, 83, 85, 88–91, 98, 101, 103–106, 111–115, 117, 120–122, 124–126, 129, 130, 132, 133, 136–140, 145

OoO Out-of-Order. 48

OS Operating System. 43, 48, 49

PIRR Predictive Interaction Remote Rendering. 5, 61, 62, 64, 67, 68, 70, 71, 76, 83, 85–87, 89–91, 93, 96, 98, 100, 103, 106, 111, 112, 114, 115, 118, 120, 123–125, 128–130, 134–136, 139, 142, 144, 145, 148, 149

PSNR Peak Signal to Noise Ratio. 27, 35, 37, 45, 46, 49, 141, 149

RTT Round-Trip Time. 69, 77, 80, 85, 90, 107

SD Send Delay. 30, 40, 65, 85, 90, 114

SL Server Latency. 18, 28, 29, 32

SSP Server-Side Prediction. 117–122, 124, 125, 128–130, 132–135, 137, 139, 140, 143, 145, 147, 148

VE Virtual Environment. 52, 53, 60, 66, 107, 149

Chapter 1. Introduction

Rendering, a computationally intensive task, is the process of automatically generating images from models (geometric representations of objects) or data sets, using computer programs. The addition of interaction allows an operator to manipulate and control/steer rendering using input devices such as a keyboard or mouse. Typical examples of these "interactive visualisation applications" are video games or visualisation software (e.g. Blender, MAYA, 3DS Max). Commonly, such applications are executed on a laptop, tablet, smartphone, etc. However, in some cases, the application resource requirements of rendering means that this is not always possible and a more powerful device is required. For example, a device may not have enough memory or compute power to perform rendering at a sufficient rate, leading users to perform worse [1, 2], experience nausea [3, 4] and lose interest [5, 2].

To mitigate the issue of resources, rendering can be offloaded to the cloud, where powerful servers receive interaction commands (e.g. from a mouse or keyboard) from a client device (laptop, tablet, smartphone, etc.), process them, perform rendering and return results (images, models, etc.) back to the client requesting them Rendering. The vast resource availability and scalability of the cloud makes Interactive Remote Rendering (IRR) systems a promising research direction, as it offers an alternative to traditional local rendering and may open markets to users and devices currently not suitable for certain tasks.

Nevertheless, IRR systems introduce some critical issues. One such issue is that of Interaction Latency (IL), which is the delay experienced, by a user, between performing an action and seeing an on-screen response/change. Although IL is present in all rendering systems, IRR systems tend to experience higher IL due to the client and rendering application being separated by a network, which introduces a delay not present where rendering is performed locally. This has led many researchers [6, 7, 5, 8, 9] to identify IL as a significant challenge.

1.1. Motivation

The last decade has seen rapid and enormous growth in the mobile device sectors, both technologically and commercially with laptops, tablets and smartphones increasingly becoming part of the average UK adult life. Smartphone ownership has grown by 61% since 2008, laptop ownership by 19% since 2009 and tablet devices by 56% since 2011. Conversely, desktop computer ownership declined from 69% in 2008 to just 28% in 2018. This 41% decrease is a

clear indication that consumers are embracing mobile technology and moving away from the traditional desktop computing environment.

Although our ability to render highly detailed scenes is increasing, 3D models are becoming more intricate as designers strive for realism and scene complexity [10]. While high quality rendering is possible with some mobile devices, smooth interaction with complex scenes consisting of millions of textured polygons is not yet possible [11]. Further, the challenges of deploying complex 3D rendering systems to mobile devices is undesirable and can lead to high energy consumption during operation [10], thereby draining batteries, generating heat and leading to other complexities.

Mobile devices, referred to as "thin clients" because of their limited resource availability (compared with "fat" desktop computers), may one day reach the same level of capability as modern desktop computers, consoles and beyond: Moore's Law dictates that computing power will double approximately every two years as transistors shrink, allowing for more to fit on a microprocessor. However, this law may be coming to an end [12], meaning that the pace at which computing power is increasing is slowing. Regardless of which device one owns, they are all, and always will be, limited in the amount of storage, memory, energy and compute power they have available. The growing scale and complexity of data and models means that mobile devices will constantly need upgrading and updating, playing a perpetual catch-up. This widening gap means that there is therefore a pressing need to find alternative solutions to visualizing and interacting with large complex sets of data and models, which are increasing in terms of dimensionality, parameter space and size [13].

The vast resource availability and scalability of the cloud makes IRR systems a promising research direction, as they offer an alternative to traditional local rendering and may open markets to users and devices currently not suitable for certain tasks.

1.2. Aims and Objectives

The aim of this thesis is to explore the challenge of IL in IRR systems, and to investigate its measurement, modelling and management.
Strategies for mitigating IL cannot be properly evaluated without the ability to accurately measure IL and the impacts of compensation techniques. The ability to accurately measure IL is therefore a crucial task. It is also important that easy-to-use techniques are developed, allowing non-experts to perform robust measurement taking and to do so with minimal impact on the wider system. Measurement techniques are investigated and developed, with focus also given to IRR systems performing interaction prediction - a scenario that introduces new complexities. However, before developing techniques for measuring latency, the ability to simulate fine-grained latency, in a controllable way, is required. This is achieved by developing a bespoke latency simulator as investigation has identified that existing solutions are not suitable.

With the ability to simulate and measure IL, a novel mitigating strategy is explored. Specifically, N-Grams, a statistical model borrowed from computational linguistics, is employed and evaluated in terms of suitability and performance. The developed model is integrated into a prediction module, which functions to predict keyboard-based user interactions ahead of time, with the overall aim being to have results delivered to the client device as near to or before the corresponding interaction has been performed. Mis-prediction will be an issue and mitigation strategies for incorrect predictions are explored.
Further, an IRR simulator platform is developed and used as a test-bed on which further experimentation is conducted. This allows for reproducible experiments in a controllable environment. A real-world IRR is then built using Unity3D with the aim of further exploring interaction prediction and its effects on IL. Towards this, the following research questions are addressed:

- RQ1. How can IL be modelled, measured and simulated? (§3)
- RQ2. Are N-Grams suitable for keyboard-based user interaction prediction? (§4)
- RQ3. How can prediction be integrated into an IRR system and what is the impact of prediction on IL? This is done with both a simulated IRR system (described in §6) and a purpose-built IRR system (described in §7).
- RQ4. Should predictions be generated on the client or on the remote server? (§9)

1.3. Methodology

Measurement approaches to IL and simulation are developed as existing tools and methods are identified to not be suitable when prediction is used. Evaluation of latency measurement is performed by comparing results with known/expected IL values. Latency simulation is validated against real-world network delays, 3 times a day (to counter broadband throttling introduced during peak-hours). An N-Gram model for predicting keyboard-based user interactions is then introduced. An IRR simulator is then developed so as to provide a stable, controllable environment for experimentation. A prediction module, designed to be separate from other system components so that it can be executed either on the client device or the server, is then developed and used to explore the effect of prediction on IL. Next, a real-world system, built using Unity3D, is introduced, where prediction is explored further. For experiments involving prediction in either the simulator or the real world IRR system, the number of steps ahead to predict ranges from 1 to 10 and the amount of interaction history used ranges from 1 to 5. Network Latency (NL) values used are 0ms, 50ms, 100ms, 200ms, 300ms and 400ms, and were chosen based on findings from literature. Managing incorrect predictions is then explored by using multiple server applications in an attempt to process additional predictions. Experiments are then performed with the prediction module on both the client and server side of the network, in an attempt to understand the differences between client-side and server-side prediction.

In the context of this research, “network latency” refers to the delay introduced, by a network, to the IRR system due to the separation of the client and server components. “Interaction latency” or simply “latency”, unless otherwise stated, refers to the total delay experienced by the user of an IRR system between performing an interaction and seeing an on-screen response.

1.4. Contributions

This thesis explores latency measurement, modelling and management in IRR systems and makes the following contributions:

1. A novel approach to measuring IL.
2. An exploration on the suitability of N-Grams for predicting keyboard-based user interactions in IRR systems.
3. A methodology for integrating prediction into IRR systems.
4. A method for calculating the time taken for IL to return to normal following an incorrect prediction.
5. An investigation into the placement of the prediction module (client-side or server-side of the network).

These contributions serve to provide developers and designers of IRR systems a robust, non-invasive method for measuring IL; help to inform as to whether N-Grams might be an appropriate prediction model; describe how prediction can be integrated into IRR systems and how to measure IL when prediction is used.

Ultimately, these contributions can be used as a framework for prediction-based IRR systems and to build more responsive IRR systems.

1.5. Thesis Outline

This thesis is organised as follows:

Chapter 2 provides some background to IRR systems. It starts off with a brief history and then describes the advantages and disadvantages of local vs remote rendering. Various Remote Rendering techniques are then introduced. Afterwards, the chapter discusses a number of latency management strategies.

Chapter 3 introduces the issue of modelling and measuring IL in IRR systems. It then describes a solution to simulating latency and how to perform measurement capture, followed by the introduction of a novel software approach to measuring IL for any IRR system. This chapter concludes with a discussion and a summary.

Chapter 4 introduces N-Grams as a potential solution to the high IL experienced in IRR systems. Assumptions and constraints are explained, as is the experimental set up and model. Two N-Gram implementations are described and evaluated and finally, a summary is presented.

Chapter 5 introduces the notion of a predictive IRR system simulator referred to as a Predictive Interaction Remote Rendering (PIRR) system. This system investigates the potential to simulate a PIRR system so as to provide an experimental test-bed for building a real-world PIRR system, and to integrate prediction.

Chapter 6 uses the PIRR simulator platform described in Chapter 5 to explore N-Gram prediction and its effect on IL with respect to how much history should be used in predictions and how far ahead prediction should be made. This chapter also describes a method for measuring the recovery rate of a PIRR system. The chapter concludes with a summary.

Chapter 7 builds on the discoveries of Chapter 6 and describes how to build a real world PIRR system using Unity3D, a popular games engine. The same experiments conducted in Chapter 6 are performed in this chapter using the Unity PIRR system, and results are compared with the simulator. Finally, a summary is provided for this chapter.

Chapter 8 looks at managing mis-predictions, which will invariably occur in a system attempting to make predictions of the future. A number of approaches are described and a potential answer to lowering the impact of incorrect predictions is proposed and investigated. A summary is then provided.

Chapter 9 introduces the notion of client-side prediction (CSP) and server-side prediction (SSP) and explores which mode benefits IL more with respect to the amount of history used in predictions and how far ahead the system predicts. A summary is provided at the end of this chapter.

Chapter 10 provides a discussion, conclusions and suggestions for future work.

Chapter 2. Background literature

Interactive computer graphics first appeared in 1963 when Ivan Sutherland (credited with being the "father of computer graphics") submitted his PhD thesis. His thesis, titled "Sketchpad: A man-machine graphical communication system", contributed many fundamental graphical computing concepts such as specialized memory structures for storing objects, the ability to zoom into images for "detailed examination" and the ability to draw straight lines [14]. This marked a turning point in human computer interaction and in 1967, Wylie et al. contributed an algorithm for hiding surfaces, giving the illusion of a three-dimensional object (a 3D model) [15]. George Romney, one of Wylie's colleagues, made significant contributions and produced the first ever render of a complex non-square object [16]. By 1982, 3D rendering was being used in film and animation and in 1989, Pixar released version one of its commercial rendering software, RenderMan, which was used to create the world's first computer animated film, Toy Story [17]. Toy Story required 117 Sun Microsystems computers, 80,0000 machine hours and produced 114,240 rendered images [18]. This was achieved using a new-at-the-time technique known as Remote Rendering (RR), where render tasks are distributed across an array of servers. For example, in Figure 2.1, animators submit render tasks to a database hosted remotely on a separate network. A collection of specialized machines (nodes) retrieve render tasks from the database, process them and produce render results in the form of images (frames). The render results are then sent to a file server where they are stored. Animators then access and retrieve the results of the render tasks from the file server. However, this feat was not possible without another emerging technology known as Cloud Computing.

Cloud Computing, the history of which can be traced back to the 1950's, is a computing paradigm enabling users to remotely access and utilize vast arrays of configurable computer system resources such as storage, memory, CPU. The term "cloud computing" was first used in 1996, in an internal document at Compaq, authored by George Favaloro [19]. Indeed, entire machines can be remotely controlled, but this was not until the 1970's when virtual machines were first conceptualized [20]. The combination of these two new technologies (distributed rendering and cloud computing) meant that for the first time, the computational complexity and resource constraints could be offloaded from local client machines to a cluster of remotely managed (and more powerful) machines.

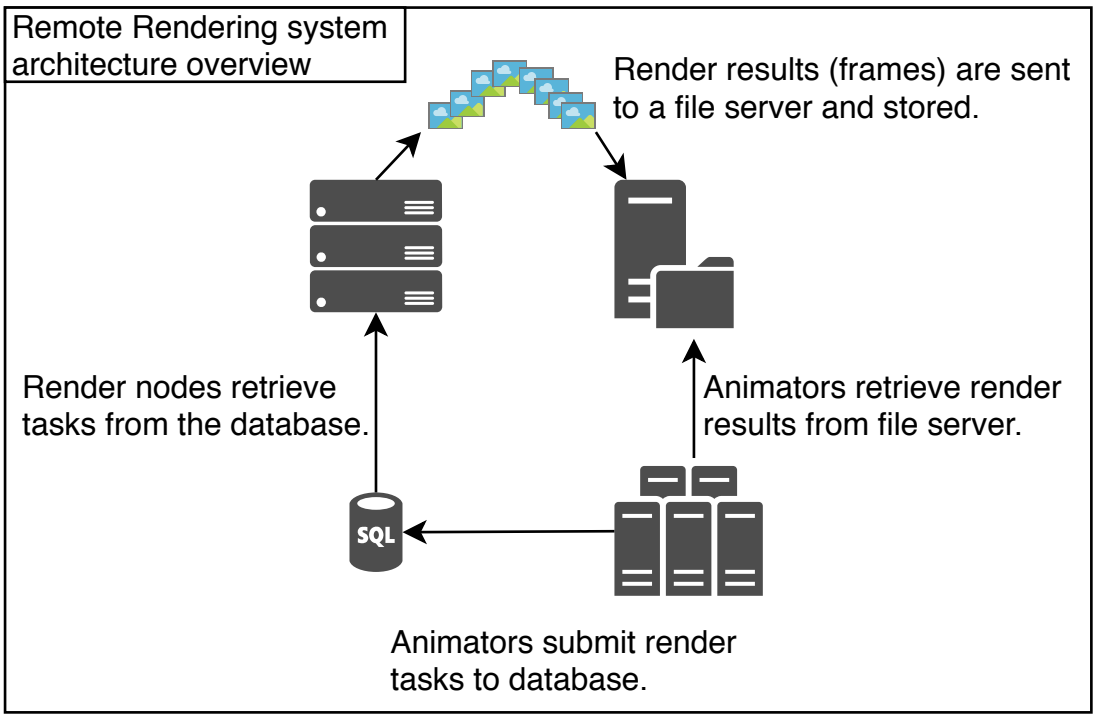


Figure 2.1 A typical non-interactive Remote Rendering system architecture.

Today, interactive computer graphics are ubiquitous, found in everything from smart watches and mobile phones, to computer games and scientific visualization. The years that followed Remote Rendering resulted in the emergence a new class of systems which have attracted a lot of attention in the last 10 years [21–24]: IRR systems enable users to interact with extremely large, complex data-sets, using a thin client device such as a smartphone, tablet or laptop. As illustrated by Figure 2.2 This is achieved by offloading the computation and resource demands from the client to the server in the Cloud. On the client, interactions from the mouse and keyboard are transmitted over a network to a remote server in the cloud. The cloud server, running a rendering application, processes received interactions, maps and applies them to the rendering engine, which produces a render result, compresses it and delivers it to the client where it is decompressed and displayed to the user.

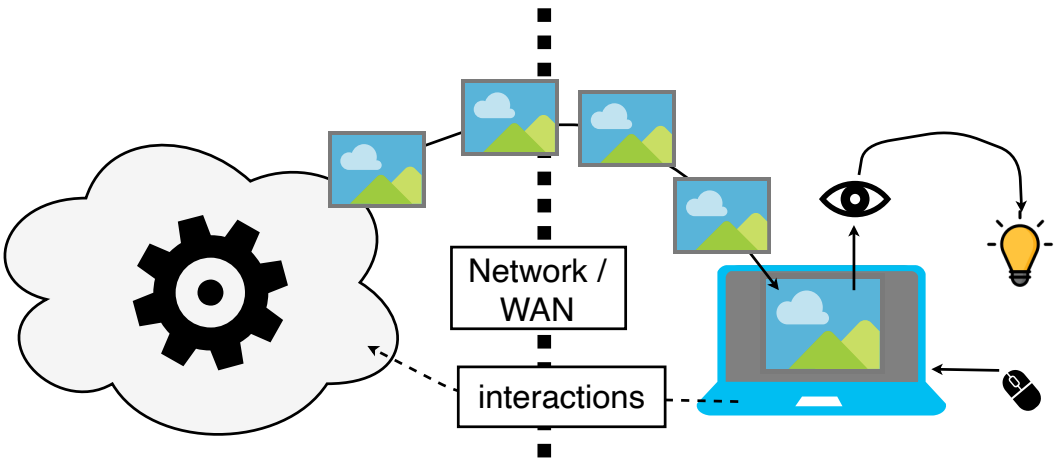


Figure 2.2 A simplistic IRR system architecture.

2.1. Local rendering vs Interactive Remote Rendering

While both local and remote visualization systems share the common goal of enabling the user to explore and interact with data, each approach has advantages and disadvantages.

2.1.1. Local rendering: advantages and disadvantages

Local interactive rendering, where all rendering and processing happens on the device local to the user, enables users to enjoy a highly responsive experience, but only when the device on which rendering is performed is sufficiently powerful enough. In these systems, a constant internet connection may not be required (for example, offline games). There is also less potential for accidentally capturing private user data since there is no need to record user interaction – they are fed directly into the rendering engine and not transmitted over a network.

However, it is difficult to test, deploy and maintain applications, especially when multiple versions are built to run on different operating systems, varying (and unknown) hardware capabilities of devices, etc. This means that diagnosing problems is very challenging, as issues may arise only on a specific device configuration. Local rendering may also consume excessive amounts of energy when in operation and devices are limited in compute power, storage capacity and rendering capabilities. All model data, 2D or 3D, is stored locally, on the device, which risks infringement of copyright laws by malicious users. Finally, hardware is expensive and therefore may limit the market reach of a rendering application.

2.1.2. Interactive remote rendering: advantages and disadvantages

IRR systems can protect sensitive and or copyright resources (model data, textures, etc.) by storing them in the cloud and not on a user device. In doing so, and by performing all rendering and processing remotely, devices can be truly thin and potentially not even require a graphics card. Users may experience state-of-the-are in rendering quality and potentially perform visualization tasks not possible on a local machine. Another advantage is that the system may be platform agnostic, not requiring different builds for different operating systems. Further, it may be easier to maintain and update the IRR system as simple client applications may only require slight modifications, while the complexity is based on a remote server.

Nevertheless, IRR systems do have their drawbacks. For instance, these systems require a constant internet connection and the reliability and broadband capacity affect user experience. The introduction of a network (the internet) increases latency, which means that systems need to be deployed as near to target users as possible. However, this may not always be possible (e.g. for geographic or legal reasons). In addition, user interactions may be unintentionally captured and transmitted over the internet, risking privacy. If the remote rendering system experiences issues (resource, power, scaling, etc.) the user experience might be affected and if the remote

server is down, the user will be unable to access applications for which they have potentially paid.

2.2. Remote rendering approaches in literature

Literature describes various Interactive Remote Rendering approaches. The following sections describe a few of the most well-known of these.

2.2.1. Model streaming

A model is a set of geometric features (mesh, textures, vertex weights, etc.) which together form a 3D representation of an object. Fundamentally, a 3D model is a digital representation of a three-dimensional object (sometimes physical and sometimes virtual). In the model streaming approach, all processing and model construction is performed on a remote machine, in the cloud. When needed, a server transmits all 3D data to the connected client for rendering.

Transmitting the entire model should only be done when a) the client has sufficient resources for storage and rendering; b) when the rendering start time is not an issue and c), when bandwidth is sufficient [8]. If the model is to be transmitted in its entirety, the client must wait until all data is received before rendering can begin. To reduce rendering start times on the client, the model can be transmitted in chunks; it may be partitioned and streamed, as demonstrated in [25] and [26]. This technique allows rendering to start on the client as soon as the first chunk is received and can offer a solution to the client not having sufficient resources, since out-dated chunks may be discarded and their memory occupied, freed.

Tang et al. assert that the transmission of models over the network increases the delay between user actions and the on-screen update, which is due to network latency and model data size [27]. The authors encountered this issue in their research and proposed a progressive streaming (see Figure 2.3) algorithm which defines two categories for interactions: 1) local browsing and 2) rotation and deformation operations. For rotation and deformation activities, the interaction is transmitted to the server as well as the local renderer. The local render computes a coarse result and displays it to the user, while the server produces a deformation at a higher Levels of Detail (LOD); The model updates are then streamed back to the client.

The fundamental issue with this approach is that rendering is performed on the client and therefore in some situations, this may not be a viable solution (e.g. the client does not have enough resources or compute power) [8]. This approach is also wasteful in terms of energy, since both the client and the server perform rendering. Another issue is that this will likely add system complexity, especially if the server needs to know the capabilities of the client requesting the model in advance. Further, if the client begins to consume additional resources, the change in resource availability may result the client no longer being capable of performing the 3D

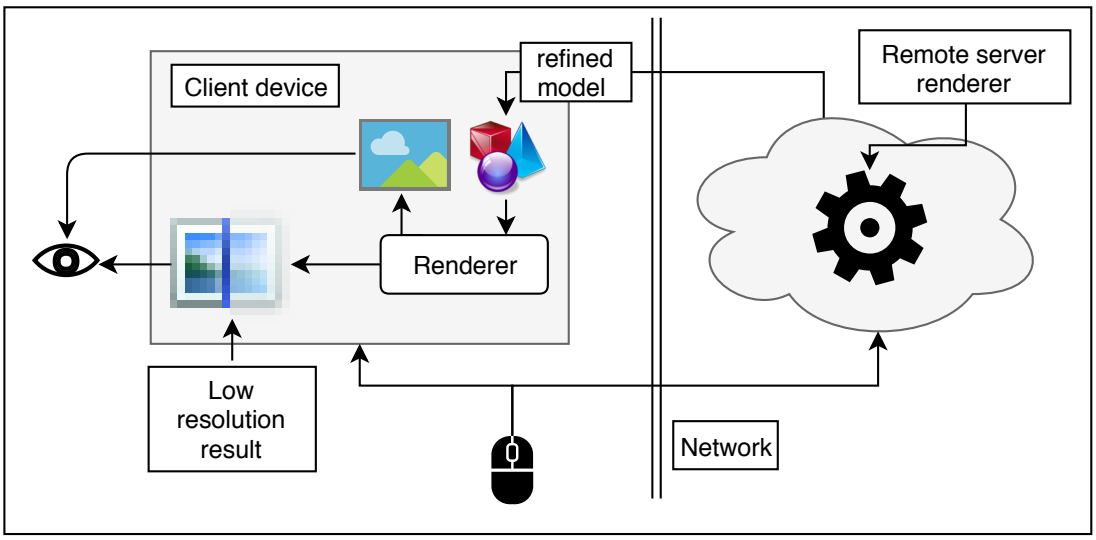


Figure 2.3 Interactive Remote Rendering system with progressive streaming.

rendering operations at interactive frame rates and in the worst-case scenario, the client may not have sufficient resources to store the 3D model. Therefore, it is likely that any such resource changes on the client need to be communicated with the server.

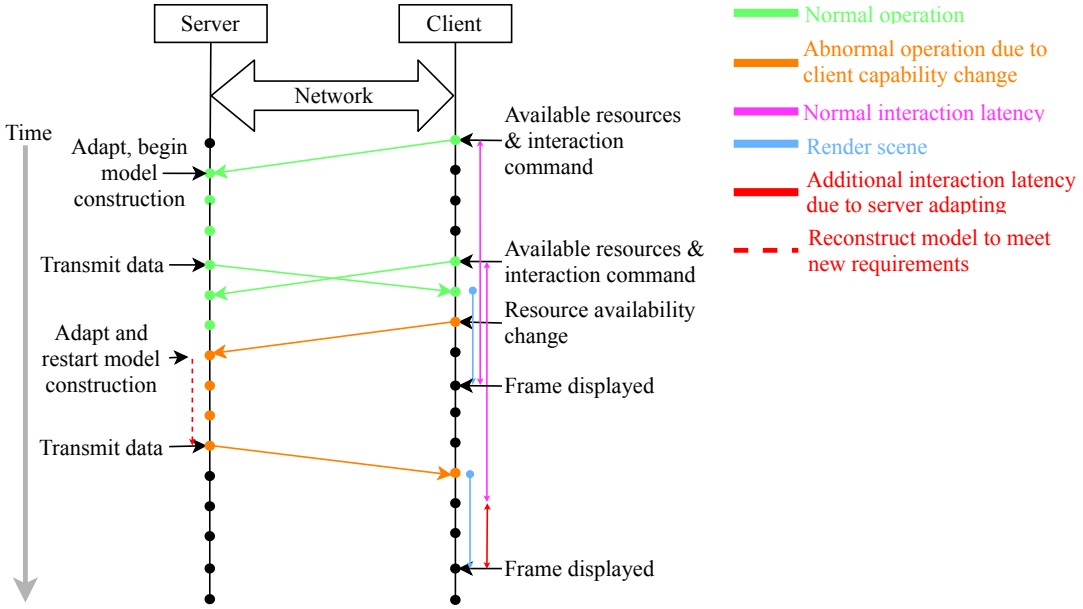


Figure 2.4 Interactive Remote Rendering system with progressive streaming.

As an example of the client-server complexities of a model streaming approach, consider Figure 2.4. The client begins by telling the server its available resources and the desired interaction command. When the server receives this information, it starts processing and model construction. When complete, model data is then transmitted to the client where it is rendered and displayed to the user (normal operation). However, if the server receives an update from the client with new constraints while model construction is taking place, the server may need to abandon the current process and restart (abnormal operation) model construction. The server

will eventually finish model construction and transmit the data to the client for rendering and display, but with added IL due to additional time spent on the server.

2.2.2. Mesh streaming

A 3D mesh is a graph: it is a collection of polygons (vertices, edges and faces) and is the surface representation of a 3D object or model. In the mesh streaming approach, the general technique is to stream data for each vertex in a predefined order to the client, where it is rendered locally on the client, while textures and other model data are streamed separately [28]. 3D meshes are commonplace in mobile games, virtual reality and many other sectors [29], however, just as in the model streaming approach, mobile devices may lack the resources required to perform certain rendering tasks [26].

Park and Lee [26] chose to focus their study on the mesh streaming approach. The authors developed a hierarchical framework for streaming large 3D meshes to mobile systems and their technique takes advantage of the fact that users cannot see the entire mesh at one time (for example, it is impossible to see all sides of a cube at the same time). Instead of streaming the entire mesh, Park and Lee partition the mesh into equally sized portions. Mesh simplification is performed on each of the partitions, which are then streamed to the client where they are rendered locally. The results of Park and Lee's study found that for the lowest resolution model, users would experience at least 120ms of IL. The bulk of this time results from transmission time of 690 vertices: 50ms.

In another example, Lin et al. demonstrate a technique for streaming meshes over a network utilizing the JPEG 2000 image compression format and for constructing a 2D representation of a static mesh [30]. For deformable meshes, such as those used in animation, Motion JPEG 2000 is used. In either case, the client initially receives a low-resolution version of the mesh to render and then receives refinement updates. Depending on the size of the complexity of the mesh, transmitting mesh data may consume large amounts of bandwidth. To reduce the amount of data to be streamed (and therefore the amount of bandwidth required), compression can be used to reduce the number of vertices on the mesh. Lossy and lossless coding techniques exist for 3D mesh compression and a good review of compression methods can be found in [31]. Other techniques such as sharing vertices [32, 26] and progressive mesh [33] streaming can also be used to reduce bandwidth, however, some techniques are only suitable for 3D meshes of low complexity [32].

2.2.3. Command streaming

In the command streaming approach, interactions are transmitted to a remote server running a 3D rendering application. When the interactions arrive on the server, the server performs processing and model transformation calculations. Once this is complete, rendering commands are generated and are then transmitted to the client where rendering is performed locally. In

order to access which rendering commands are required, they can be intercepted using a technique known as API hooking. For DirectX and OpenGL, this can be achieved with a well-known library called Detours [34].

Eisert and Fechteler describe such a framework in [35]. The authors point out that the encoding of commands is independent from image resolution, which means that high-resolution output can be achieved on the client. However, they also report that bit rates are not very predictable and high data rate peaks can be expected. They further state that, crucially, some graphics command calls require feedback. Since there is no way of knowing what must be done with the return values, and the authors admit that this approach can introduce unacceptable round-trip delays. To mitigate this issue, Eisert and Fechteler simulated the graphics card state of the client on the server so that they can reply directly within the server environment, rather than having to communicate feedback from the client. Of course, as the authors indicate, with this approach, rendering is still performed locally, and so not all applications can be supported.

2.2.4. In-situ approach

In the scientific community, data visualization plays a crucial role in the discovery process. Typically, post-processing techniques are employed where a simulation is performed (often on remote servers or a supercomputer) and the data results are written to disk. When required, data results can be retrieved by being read from disk into a visualization program for exploration. Unfortunately, scientific data from simulations (used here to describe the process of performing a computer experiment which results in a data set(s)) is growing in quantity, quality, and with some data sets hundreds or in millions of Gigabytes in size [36]. Our ability to process, manipulate and visualize these data sets is being challenged, because the gap between being able to produce data, and perform IO (Input Output) operations is widening [21, 37, 38]. Reducing the need to read and write from disk is highly desirable as it is a known bottleneck to High Performance Computing (HPC) [37, 39].

The in-situ approach attempts to eliminate the need for expensive (energy, time/speed) disk operations as it aims to enable the operator to visualize the data as it is produced by the running simulation [40]. Simulations can take days, weeks or months to complete and be ready for visualization. The in-situ approach mitigates the risk of having to restart a simulation (either during its execution or after completion due to errors) and allows the researcher to “steer” the simulation as it develops. Researchers can query, visualize and watch the simulation evolve, giving the added advantage of making “accidental” discoveries or steer it in a different direction when and if needed [36].

The results of a simulation can be stored on the file system which can be used for visualization at a later point in time. However, as explained earlier, disk I/O operations are expensive and slow. Ahrens et al. [21] developed an image-based framework for in-situ visualization on top of a

well-known open source project called ParaView. To reduce the cost of disk I/O operations the authors developed a custom image database which is fed image results as the simulation executes. The end user can access the image database through a user interface that allows for two modes of interaction (animation and selection). To enable querying, the authors saved meta data describing how and when images were generated. The meta data and images are stored in the custom image database. Using meta data, queries can be constructed for image compositing, which can be performed to produce new visualizations. The authors managed to achieve frame rates of 12 fps. While impressive given the scale of the data (approximately 24TBs), frame rates still fall well below the desirable 30fps for smooth interaction. Unfortunately, their approach was based on local network conditions and so network latency, a critical factor in IRR systems, was not considered. Finally, often, due to the size and volume of the output images, storage on local devices is simply impossible and requires distributed image servers. Many techniques such as co-processing, concurrent processing and a hybrid mix combining both co and concurrent processing exist for in-situ visualization approaches and for a more in-depth look, see [36, 21, 38].

2.2.5. Hybrid approaches

The hybrid approach is a mix of local and cloud server rendering. There are a number of hybrid approaches available in the public domain. For example, a well-known open source project, VirtualGL uses this technique, and was started in the 2000's as a solution to the growing demand of running complex 3D applications on thin client devices. VirtualGL uses the command streaming approach, allowing for 3D operations to be redirected to a remote server hosting GPUs and other resources. All 2D operations are allowed to continue as normal on the client. VirtualGL dubbed this technique "GLX Forking" [41]. VisIt, which is now over 14 years old, is another example [38]. It has grown to be adaptable in that it can utilize either the model or image streaming (discussed in the next section) approach: if the client has insufficient resources or graphics capabilities, the image streaming approach is employed, otherwise, the model streaming approach is adopted to increase interactivity performance. VisIt is also highly scalable and extensible. Lastly, Visapult is another example of a hybrid approach. Visapult performs partial rendering on a remote server and then transmits the partial results to the client, where rendering is completed. Its main focus is on "remote and distributed, high performance direct volume rendering" [42].

2.2.6. Image streaming approach

The image streaming approach has been around since the 1990s [43]. Contrary to other approaches, such as command or model streaming, this approach transmits only images to the client [22]. All processing, model construction and rendering is performed on a remote server in the cloud. When a user performs an action, such as a mouse click, that action is sent to the remote server which maps the received command to a model, applies a transformation and

renders the scene. Once rendering is complete, the resulting image is then transmitted back to the client where it is displayed to the user (see 2.2).

Moreland et al. explored a technique for remote rendering of ultra-scale data in [44]. They proposed a simple client-server architecture: the server performs all rendering and delivers the rendering results to an image cache on the client. When interactions are performed on the client, camera coordinates are sent over the network to the server. Meanwhile, a lookup is performed in the cache. The cached images are used to synthesize images at novel viewpoints. The authors state that their technique "will provide an interactive rendering experience regardless of network performance", although they do not provide any measurements regarding IL or frame rate.

Sterk and Palacio described a remote rendering solution for terrain visualization on mobile devices where the authors compared the remote rendering approach with local rendering, but in order to do so, they had to scale down the geographical data [7]. Their findings lead to the conclusion that rendering remotely provides the best visual quality and is "probably the only option for GPU-less devices". However, the authors also note that the remote rendering system requires a constant internet connection and is less responsive than local rendering, resulting in IL of 4.5 seconds (IL for local rendering is not reported).

Wessels et al. proposed a framework for remote rendering using websockets [13]. Their framework consisted of a server, visualization engine, a daemon and a client. With the visualization located on the server, interactions are received from the client and processed via the daemon. In addition to acting as the communications gateway between the client and the server, the daemon is also responsible for launching the visualization process and managing client connections. On the client, frames received are unencoded from base64 and displayed to the user. No IL measurements are reported.

Transmitting images over the network is expensive in terms of bandwidth. The higher the resolution of the image, the larger the transmission times – which results in larger IL. As such, scheduling and rate control mechanisms may be used to adjust the server to match the client's ability to download and display images [22, 8]. Fortunately, this is facilitated by the fact that bandwidth usage is predictable and bounded for the image based approach, as image dimensions, data size and frame rate can be communicated in advance [40].

The image streaming approach can be far less complex than others such as the model streaming approach. The reason for this is that only images are ever transmitted to clients: thin client devices may enjoy the experience of complex, resource-hungry (GPU, CPU, RAM, HDD) applications without the need for specialized hardware and/or software [13, 7]. No rendering needs to take place and so the client does not even need a GPU; only the ability to display an image and communicate with the server is required. Collaboration may be easier to implement

since updates sent to connected clients are images; no scene updates, positional data or states need to be synchronised. Additionally, this approach provides a cross-platform remote rendering solution as they do not have to be programmed for specific operating systems, such as Windows or Android. Maintenance is likely to be easier too: client devices need only a web browser, which is already available from multiple vendors; there are no drivers or software updates to consider on the client device (unless a specially designed client-viewer is required, otherwise a browser is sufficient), and end users do not have to constantly upgrade their devices to meet the demanding specifications of the applications that they wish to run. The IRR provider only needs to maintain the software and hardware of the server. Finally, Intellectual Property Rights (IPR) are protected and data as models remain safely on the cloud server as was demonstrated in [45] and in [46].

2.2.7. Summary of approaches

The model, mesh and command streaming approaches are well suited to situations where the client has sufficient resources to render at interactive frame rates. The downside with these approaches is that rendering is performed locally, on the client. In addition, they are complex in that they require the server to know the resource and compute capabilities of the client [8], and need to receive periodic updates from the client in case of resource availability changes. In addition to this, potentially copyrighted material (3D models, textures, meta, etc.) must be sent across the internet to possibly unknown and untrusted clients.

The in-situ approach is typically used when visualizing the output from scientific simulations. As the simulation executes, images are generated and presented to the user. Image results can be written to disk, or saved to a database, but disk space and IO must be considered. With some simulations, the output results or images may require too much diskspace for this approach to be practical. Further, when performed over a network, latency will impact responsiveness and depending on image resolution, may consume excessive bandwidth.

Of all the approaches to IRR, the image streaming approach has the potential to have the greatest impact on visualization. However, the large amount of bandwidth required as well as the long network latency delays are potential issues. To reduce the burden of multiple images being streamed to the client, a panorama can be constructed and streamed to the connected client. Alternately, a single virtual camera can be used and transformed to different viewing angles or, multiple virtual cameras can be used, as shown in [22]. Nevertheless, techniques such as the image streaming approach offer a potentially compelling research direction, providing issues surrounding latency are addressed. The ability to interact with complex 3D rendering applications on any thin, internet connected device, is an attractive proposition. The approach offers end users the ability to play the latest games, run the most computationally and resource demanding 3D rendering applications and interact with the most complex visualizations without the need to constantly upgrade their devices. Developers will find that maintaining 3D rendering

applications for a single, known system environment is far easier than the current situation where client devices are unknown with different OS's, different CPU's, RAM, GPU's and storage capabilities.

Table 2.1 IRR Approaches and their pros and cons.

Model streaming	
Pros	+ Low interaction latency
Cons	- Client may not have sufficient storage capacity to accommodate model data. - Client needs specialized hardware (GPU). - Client must be capable of rendering at interactive frame rates (30FPS) - Potential for data theft. - Difficult to load balance in terms of bandwidth. - Difficult to update clients which require different updates for different operating systems.
Command streaming	
Pros	+ Low interaction latency. + Adaptable for almost any application (closed or open source)
Cons	- system may be unstable as API hooking can result in application crashes. - Client may not have sufficient storage capacity to accommodate model data. - Client needs specialized hardware (GPU). - Client must be capable of rendering at interactive frame rates (30FPS). - Potential for data theft. - Difficult to load balance in terms of bandwidth. - Difficult to update clients which require different updates for different operating systems.
In-situ	
Pros	+ Well suited to big complex data sets such as those in scientific simulations. + Allows "steering" of the simulation. + Removes need for expensive disk I/O operations.
Cons	- Requires close collaboration between application developers, visualization scientists and researchers; not general purpose. - Produces large volumes of output data.
Image based	
Pros	+ Client can be truly thin with no specialize hardware. + Data not,exposed to client and is therefore safe from theft. + Bandwidth usage is, predictable and bounded, making load balancing easy. easy to update, clients. + Platform agnostic.
Cons	- High interaction latency. - High bandwidth consumption.
Hybrid	
Pros	+ Potentially low interaction latency.
Cons	- Complex application. - Difficult to maintain. - Usually not general-purpose solution.

2.3. Managing Latency

Interaction Latency (IL) is the difference in time between the moment an interaction is registered by the client device, and the point at which the corresponding frame result is displayed to the user. Latency can occur at various locations within an IRR system; for example, on the client device such as during the decoding stage, within the rendering pipeline on the server, or during client-server communication. IL is the primary issue of IRR systems [5]. It is defined by:

$$IL = SL + CL + 2NL \tag{2.1}$$

Where Server Latency (SL) is the total delay on the server, Client Latency (CL) the total delay on the client and NL is the Network Latency.

Studies have shown that users are able to detect latency as low as 2.38ms [47] and that users will interpret an action as the cause of an event, when latency is less than 70ms. When latency exceeds 160ms, users experience a disconnect between the event and the action that caused it [48], possibly leading the user to become confused. High latency also results in less engagement: players of online games have been shown to play far less when experiencing delays in excess of 250ms, compared to those who experienced latency of 150ms [49]. Thus, as little as 100ms latency can be responsible for a loss of up to 75% user engagement; a significant figure for companies such as Blizzard who own the Massively Multiplayer Online (MMO) franchise, World of Warcraft.

Importantly, not all systems share the same sensitivity to latency. For example, in the popular first-person shooter game "Unreal Tournament 2003", latency less that 75ms is acceptable, while latency above that can result in user accuracy and score reducing by up to 50% [50]. Claypool [51] analysed the effect of latency on a real-time strategy game, Warcraft 3, and concludes that delays ranging from hundreds of seconds to several seconds do not significantly affect user performance and the same would apply to turn-based games. Interestingly, Claypool also highlights the fact that different tasks in games such as combat or building are more sensitive to latency than other tasks; this is likely to be true for other 3D rendering applications such as virtual environment navigation or training simulators.

Virtual Reality (VR) and Mixed Reality (MR) are highly sensitive to latency. Zheng et al. [21] notes that a NASA study concludes that latencies need to be less than 2.5ms for head movements of more than 100 degrees in Head Mounted Displays (HMD). Latencies exceeding certain thresholds can also lead users to experience nausea, commonly called simulator sickness in VR systems. Presently, VR/MR devices are designed as specialized platforms with special attention given to latency. Reducing this latency is critical if interaction with large, complex data sets in a VR/MR environment is desired. Additionally, since VR and MR are predicted to generate a combined revenue of $120 billion by 2020 [52] and that high latency severely affects user engagement, there is great value in paying close attention to latency concerns when designing systems sensitive to it. Further, IL is also a concern for exploratory visual analysis. For example, Liu et al. show that an IL greater than 500ms results in decreased user activity, poorer data set coverage and reduces rates of observation, generalization and hypothesis formation [5].

A combination of factors impact IL, with NL, being one of the most significant contributors. Networks are not always stable and can cause available bandwidth to drop, reducing the rate at which data is downloaded to the client. Similarly, wireless networks may lose signal, causing significant delays. Unfortunately, NL is inherent in all network distributed applications and is

dictated by various factors such as the physical distance between the client and server machines, and the type of material (i.e. copper vs fibre) used to transfer data across the network [53]. The most common approach to dealing with IL is to simply have servers located as near to their target users as possible. Unfortunately, this is not always possible as geographic, economic and ethical constraints may be prohibitive. The next sections describe various ways of managing IL.

2.3.1. Level of detail (LOD) management

Despite massive advances in graphics hardware, our ability to render complex 3D models at interactive frame rates is still a significant challenge to graphics programmers. The growth in terms of size and complexity of 3D models appears to outpace our development of faster, more powerful rendering hardware. Thus, graphics programmers have to constantly make choices of trade-off between realism (complex, detailed models) and performance (interactive frame rates, smoothness of movements and responsiveness).

LOD management is a technique for controlling the complexity of a 3D model by generating a visual approximation and is performed after user input is sampled. By reducing the vertex count, items further away from the scene camera can be rendered with a lower resolution, as much detail is not visible from large distances. Conversely, LOD management can also be used to increase the level of detail so that as a scene camera moves closer to an object, higher resolution models are generated. A simple example of this is looking at a mountain from the window: you cannot see the leaves on the trees until you get closer. Figure 2.5 illustrates a simple example of different LODs: (A) 9,902 vertices for maximum detail close- ups; (B) 2,452 vertices; (C) 602 vertices; (D) 134 vertices for low detail objects far away. As such, LOD management is an excellent technique for reducing model complexity and rendering time, as well as increasing frame rates [54]. Additionally, by reducing model complexity, the physical storage size of the model is reduced too, making transmission of such models require less bandwidth.

In systems where latency is critical, model fidelity can be sacrificed so that the transmission payload is reduced, resulting in lower IL in cases where the entire model must be transmitted before rendering can commence. The side effect is that by reducing the LOD of a model, the resulting rendered image quality is reduced. However, such effects are unlikely to be noticed or cause discomfort when viewed from sufficient distances, or when moving at sufficient speeds through a 3D scene.

For an excellent and in-depth discussion, see Level of Detail for 3D Graphics [54].

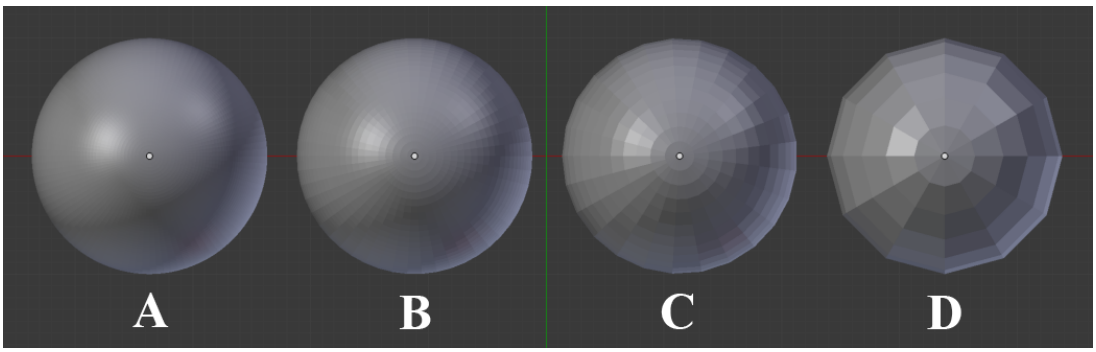


Figure 2.5 A sphere presented at various Level of Detail (LOD).

2.3.2. Image warping

3D Image warping is an Image Based Rendering (IBR) algorithm that uses one or more images with depth information at a given viewing angle to synthesize the same scene from a different point of view [55]. Once an interaction occurs, the image warping algorithm executes and produces the newly warped image. The result is the decoupling of rendering from interaction and can be used for latency reduction because it reduces IL to the time it takes for the algorithm to execute. An example is shown in Figure 2.6, where the image of a hedgehog (left) has been warped to a new perspective of approximately 30 degrees (right) and as a result, information has been lost and areas are therefore void of data (blue)

Figure 2.6 An example of image warping. Approximately 30 degrees warp.

Image warping seems to have first appeared more than 30 years ago and an early example of its use is when researchers at Carnegie Mellon, USA, built a machine called “WARP”, which was designed as a “programmable systolic array machine” and was used to perform mapping operations [56]. In one example, WARP was used to remap a distorted perspective projection from a camera mounted on the roof a robot driving down a road, to a flat, 2D perspective (aerial view) so that image processing techniques could be applied, and the road edges identified.

Since then, 3D image warping techniques have been proposed with the aim of being able to generate images at novel viewpoints with depth. As such, 3D image warping is often referred to

as Depth Image Based Rendering (DIBR) in literature. The algorithm takes the following input parameters:

1. Source viewpoint consisting of camera position, orientation, focal distance, horizontal and vertical field of views
2. A colour buffer, depth buffer and a combined model-view-projection matrix.
3. Target viewpoint consisting of camera position, orientation, focal distance, horizontal and vertical field of views

The algorithm output result is a single image, warped to a novel view (the target view) . Used in an IRR context, this algorithm can execute on the client and be used to synthesise images (in response to interactions) between frames arriving from the server and therefore compensates system latency [55], although literature does not describe to which extent this is possible.

Shi [57] proposed an IRR platform for mobile clients using 3D image warping. On the server, multiple depth images are generated from user input arriving from the client. The depth images are sent across the network to the client which then runs the 3D image warping algorithm. An issue he found was that generating a depth image at multiple viewpoints is computationally expensive and time consuming. Additionally, each reference image needs to be transmitted with its corresponding colour and depth information, requiring extra bandwidth. To reduce the workload and required bandwidth, viewpoint selection is performed by selecting those reference images that are most similar to the desired target viewpoint. The reference images which most closely match the desired output image can then be computed by iterating over each of them and for each one, compute the viewpoint similarity between the reference viewpoint and the target viewpoint. Further information on the algorithm can be obtained in and [58, 55, 59].

A common problem with 3D image warping is that of hole artefacts [60] (also known as exposure): regions normally hidden by foreground objects (in the reference image) are made visible in the new synthesized view. Hole artefacts arise because there is not enough knowledge of the scene contained in the reference images to completely synthesize the scene for a new viewpoint. Double warping [9, 55], where two reference images are warped and then composited, can be used to compensate for the hole artefacts, but does not always yield satisfactory results. An example of how hole artefacts arise is illustrated in Figure 2.7.

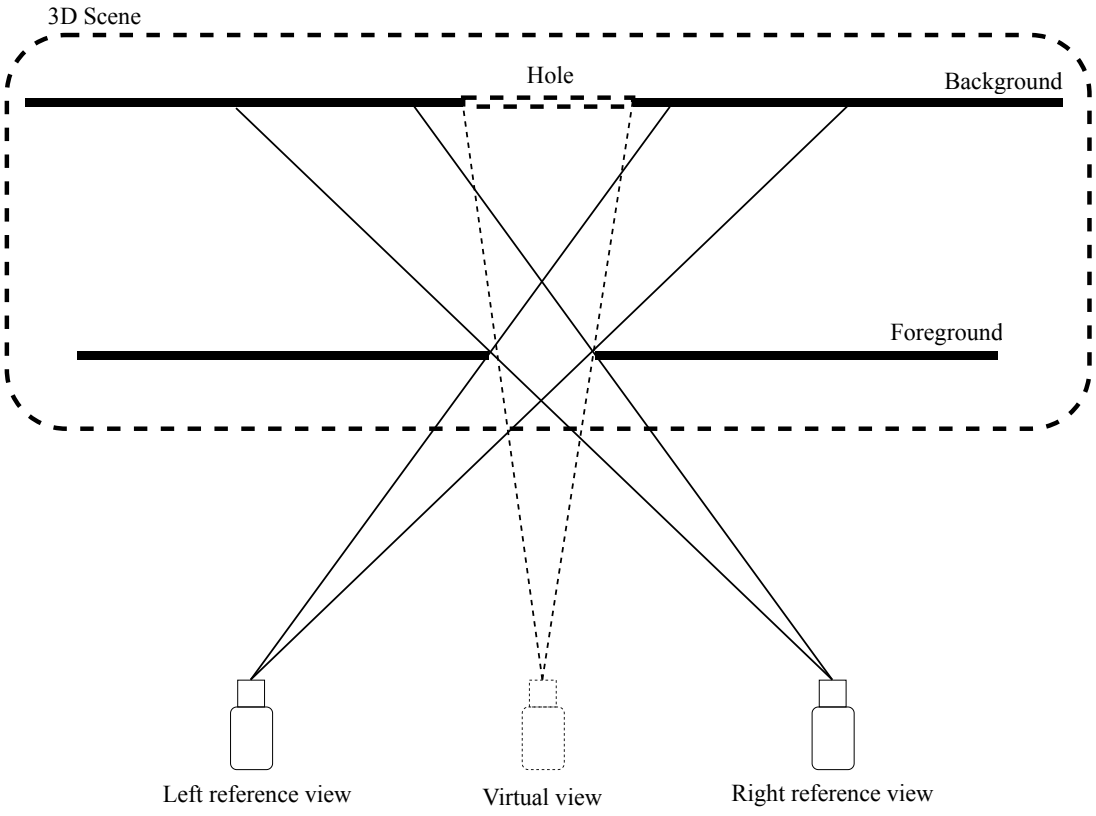


Figure 2.7 An explanation of hole artefact caused by image warping. Image source [61]

At first, 3D image warping might appear to be an excellent way to reduce interaction latency, but serious problems exist. As noted by Shi [9, 57], 3D image warping not only results in hole artefacts, but fails for certain special effects such as shadows and animation; it is not suited for scenes with "moving foreground objects that are not controlled by the rendering viewpoint". Thus, [57] recommends that 3D image warping be designed and integrated with the original application to mitigate such situations.

2.3.3. Prefetch

In IRR systems, data (image data, model data, etc.) are transmitted from the server to the client where they are presented to the user. The transmission of this data from the server to the client results in NL, adding to the overall IL. In video streaming services such as YouTube and Netflix, buffering techniques are used because the stream consists of sequential known images, known as a priori [22]. Buffering allows the streaming service to deliver content to the client before it is needed. Not only does buffering provide smooth playback, but it also compensates for unstable network connections by acting as a sort of "suspension" where a sufficient number of buffered frames provides time for the system to recover from network delays. Additionally, since the stream consists of images, network requirements are predictable and bounded: the framerate and size of the data can be used to inform how much data must be downloaded and at which rate.

In IRR systems, the user controls the direction of the visualisation or simulation. Typically, a user will perform an action on the client and sometime later, the client will receive data

corresponding to the action performed from the server. Instead of the server waiting for the interaction commands from the client, the server can produce the data required ahead of time and send that data to the client. Therefore, the client does not need to wait for a response from the server, reducing perceived NL and IL.

As an example of prefetch, consider a virtual walkthrough hosted on a remote server, such as in Google Street View, where a user controls the virtual camera from a client device (his/her laptop). In this example, the interaction and environment are constrained such that the user can only move in three directions: forward, left and right. As the user navigates forwards through the virtual walkthrough, the server generates the data (image, model, etc.) for not just one viewpoint, but several (three in this example) viewpoints. That data is transmitted to the client where it is stored in a buffer or a local cache. If the user turns to the left, to the right or forward, the client does not need to request data from the server as the required data has been already been acquired.

One attempt to take advantage of the properties of buffering can be found in [22], where Boukerche et. al. propose using a remote rendering solution for interacting with virtual environment (VE) applications, and contribute a scheduling and buffering mechanism [22] combined with a simple look-ahead strategy. In [62], Higgins et al introduce a prototype called "IMP", which aims to hide the complexities of prefetching decisions on mobile devices. An obvious problem with prefetch is that if the transmitted viewpoint data does not accurately match the corresponding interaction issued by the client, the user will be displayed a frame that does not depict the expected scene transformation. One could argue that the server could generate and transmit data for many more viewpoints to the client and that the client could then select the appropriate viewpoint data to display to the user. However, blindly sending data for an arbitrary number of viewpoints is inefficient as out of all the viewpoints transmitted, only one will be used, which would result in large bandwidth consumption.

There are two approaches to prefetch: brute force, where data for multiple viewpoints are transmitted to the client, and prediction, where historic interaction data is modelled so that only the most likely required viewpoint data is transmitted.

2.3.4. Brute force based prefetch

In the brute force approach, the server sends data corresponding to multiple viewpoints to the client. If bandwidth was unlimited, the brute force approach would seem appealing at first because the data for every possible viewpoint could be transmitted to the client. However, even if the client has data for every possible viewpoint there is still a problem: selection. The larger the number of viewpoints for the client to choose from, the longer the selection process. Reducing the number of prefetched viewpoint data is an option, but if the prefetched data does

not contain the viewpoint required by the client, IL will be experienced while waiting for the server to transmit further viewpoint data.

2.3.5. Prediction

Prediction can be a powerful tool for latency hiding. For instance, user interactions can be modelled and predicted, which can then be used to generate future frames that can be transmitted to the client before the user requires them.

In [63, 22] Boukerche and Pazzi use a client-server architecture where they first construct (on the server) a set of 12 images based on user position and direction in a virtual environment. The server then warps the images to cylindrical coordinates. Next, a simple prediction technique is employed to assign priorities to the images. The images are then inserted into a buffer according to their priority (highest priority first) and are then transmitted to the client. The client receives the 12 images from the server which it uses to construct a panoramic view of the virtual world. As the user moves forward within the virtual scene, the panorama being displayed is zoomed to a threshold [63], allowing the user to perceive motion without visible image quality degradation. As this happens, the next panorama is constructed and pushed into the buffer, ready to be streamed to the client. The design proposed by Boukerche and Pazzi is limited in that user movements have been severely constrained so as to reduce bandwidth and to simplify the prediction process. Further, no IL measurements are provided.

Zhou et al. [64] propose an algorithm for prefetching content within a 3D scene based on user access patterns. The authors argue that by studying the access patterns of scene content, inference can be made as to which resources will be required in the near future, and that they can be fetched from the server, delivered to the client and pre-rendered, before the user client devices requires them.

Dead Reckoning (DR), a technique born out of the Distributed Interactive Simulation (DIS) protocol, which was developed by the U.S military [65], has been used to hide NL and to reduce bandwidth: To hide latency, all connected clients agree in advance upon a set of algorithms that will be used to predict the future state of in-application entities. Clients apply the predicted states when required, rather than waiting for updates from the server. When predictions errors cross a threshold, a correction is issued. To reduce bandwidth, DR can be used to minimize the number of messages required to be sent between the connected clients and server [66].

$$P = P_o + V_o\Delta t + \frac{1}{2}A\Delta t^2 \tag{2.2}$$

Where P is the predicted position, V_o is the velocity, A is acceleration and Δt^2 is time difference , between the present and the last update.

DR is often used in "Authorative server" architectures, where the client performs prediction and rendering locally. Rather than wait for results from the server, the client application processes local inputs and generates "predicted" future states, effectively running ahead of the server application. This is achieved by allowing the client to assume that its inputs will be accepted by the server. However, when the server rejects an input, the client is "reset" to the last know correct state. This is challenging to correct as the client is running a future state of the simulation, and any corrections received from the server are for a past state, meaning that the client application must calculate the correct state from the correction point, up to the present point in time. This causes undesirable "jerks" of the image on-screen and is sometimes referred to as "rubber banding" by the video game community. Nevertheless, this negative effect can be lessened by smoothing errors [67].

While DR works well for small latencies, large latencies can result is significant errors. Another issue is that the prediction is linear, which is well suited to predicting positions along a straight trajectory but fails to predict changes in direction.

Lazem et al. proposed an prediction-based prefetching scheme for VE's [68]. The authors proposed a linear model for movement prediction, predicting one step ahead since any prediction errors would propagate with further prediction. Unfortunately, the authors did not report on IL measurements. Similarly, Chan et al. proposed a "hybrid motion prediction method for caching and prefetching" algorithm for distributed VE's [66]. The authors report that their technique works well for predicting mouse movements with both single and multiple steps ahead and that in some "typical navigation patterns", performance is increased significantly. Again, the authors failed to describe IL measurements.

Although previous user actions can be modelled and applied to the prediction algorithm, it can be very difficult to do so given the number of possible key/mouse inputs. Exactly which actions should be modelled depends on the purpose of the IRR system. For example, in a virtual walk-through environment, user movements such as forward, back, left and right can be modelled easily. In a flight simulator, the movements can be increased from 4 possible directions to 6 as the user cannot move backwards, but can control pitch (up and down), yaw (left and right) and roll (rotation along the longitudinal axis). Yet, in more dynamic environments such as in video games, user movements are unconstrained: the user can control movements for forward, back, left, right, pitch and yaw. Moreover, in video games such as a first-person shooter, users have a range of interactions that can be performed (e.g. firing a weapon). As noted by Chu et al. [24], modelling all possible actions can lead to excessive IL due to a state space explosion. The authors identify two main interaction classes and builds "speculation engines" for each of them. The first is navigation (forward, left, right, etc.) and the second is impulse (e.g. firing a weapon). To compensate for the many combinations of user actions, they used state space sub-sampling and event stream time shifting. To forecast future user inputs, a discrete Markov chain was

selected for use, as the results were comparable to that of neural networks, linear and polynomial regression models. Chu et al. demonstrated that with their technique, IL can be halved. However, while the authors claim that their system is capable of masking up to 250ms of IL, their approach was tested only on video games and so it remains to be seen if their method can be adopted for other IRR systems, such as those designed to visualize large data sets.

2.4. Summary

This chapter has explored the various approaches to IRR and managing IL, as described in literature. While there are a number of approaches to performing IRR, they all suffer IL larger than would normally be experienced during local rendering. As described, there are a number of strategies with which IL can be reduced. However, current approaches are not sufficient as they either require the transmission of (potentially sensitive/valuable) model data over a network, require that all model data be received by the client before rendering can be performed, are overly complex, result in significant image artefacts, cause "jerking motion" to be experienced by the user of the client and many other issues. In addition, it is frequent that IL measurements are not provided by the described literature, making it difficult to compare approaches.
From the literature, it is also clear that there is a lack of discussion on the modelling and simulation of IL in IRR systems, as well as on its measurement. In the next chapter, these topics are explored further.

Chapter 3. Measuring and Simulating Latency in Interactive Remote Rendering Systems

3.1. Overview

Measuring IL in IRR systems [40], is a crucial yet challenging task. IL is typically not controllable, making it difficult to study compensation techniques, their effects on user performance and their impact on user experience. Additionally, current measurement techniques are not general-purpose enough, which further complicates system performance bench-marking. In the literature, most results presented fail to describe a) how IL is measured and b) what the measured IL is (if measured). There therefore appears to be a lack of understanding of how best to perform IL measurement in IRR systems. Moreover, before IL can be measured, it must be modelled and a controllable delay must be present.

This chapter therefore aims to address RQ1:

1. How is IL modelled and simulated?
2. How can IL be measured?

Towards this, IL is first modelled (§3.2) so as to understand its sources. A latency simulator is then built so that controllable delays can be inserted into future experiments which aim to mitigate the effects of IL (§3.3). Afterwards, a new Latency Measurement Tool (LMT) is developed (§3.5), which captures the image output of applications and uses the Peak Signal to Noise Ratio (PSNR) between image-pairs to determine when an interaction has occurred.
A rudimentary IRR system is developed and the latency simulator is integrated into it. This provided a platform on which to evaluate both the latency simulator and the introduced approach to IL measurement. To evaluate the latency simulator, results are compared with measurements collected from a real-world network. To validate the introduced measurement approach, a simple Windows Form application (§3.5.1) is initially developed to determine base-line measurements of the introduced measurement technique. Results are then collected from the IRR system by using the LMT, while running both over WAN and using varying degrees of simulated latency.

3.2. Modelling latency

The main challenge in measuring IL is the need to identify the exact moment an interaction occurs, as well as when the corresponding frame update is visible to the user. This is difficult

since in rendering systems, frame updates are typically the result of both user interaction and scene mechanics. For example, in a 3D racing game, an AI-controlled car may overtake a player independently of the user interactions; despite performing no action, frames independent of user input will continue to arrive from the server and be displayed on the client. In order to develop techniques to measure IL, latency sources must be identified.

Towards this, we identify five main sources of IL in typical IRR systems:

Input Device Latency (IDL) is the delay contributed by the input device, usually an external controller. This delay is the difference in time taken between an electronic signal (due to a button being pressed, for example) being generated on the hardware and having the input processed (memory updated in the OS event buffer). This delay may increase due to poor wireless connectivity, poorly charged batteries, etc.

Client Latency (CL) is the total delay experienced on the client device. Since CL is the result of latency introduced both before interactions leave the client and after image results arrive, we identify that CL is composed of two sub-latencies: CL_1 and CL_2. CL_1 is the delay from the point at which an interaction is received on the client, until that interaction exits the client, while CL_2 is the delay between the client receiving a response from the server and processing (e.g. image decompression, client state updates) it, up to the point at which the pixels are pushed to the display.

Network Latency (NL) is the total delay caused by the transmission of a message between the client and the server. NL occurs in two directions: from the client to the server (NL_{up}) and from the server back to the client (NL_{down}). The physical distance between the client and server is the primary source of NL. However, network routing, congestion and delivery (copper wire vs fibre optic, for example) all impact NL.

Server Latency (SL) is the total delay experienced on the server, from the point at which the interaction message is received, until a response message is generated and has left the server. SL will always be present, however, poor rendering speeds, coding inefficiencies, image compression, low-end hardware, etc., will add to the total SL and therefore negatively impact IL.

Display Latency (DL) exists in all monitors. It is the delay experienced between the display system receiving an input signal and the image being displayed on the screen. DL varies across display systems and is a combination of video input latency (time taken to perform up-scaling and smoothing and double buffer switching, if any) and pixel response time (the time taken for a physical pixel to update).
A general IRR pipeline model of the above latency sources is illustrated in Figure 3.1. To illustrate further, consider an interaction occurring at t_0. At t_1 the interaction event is raised by

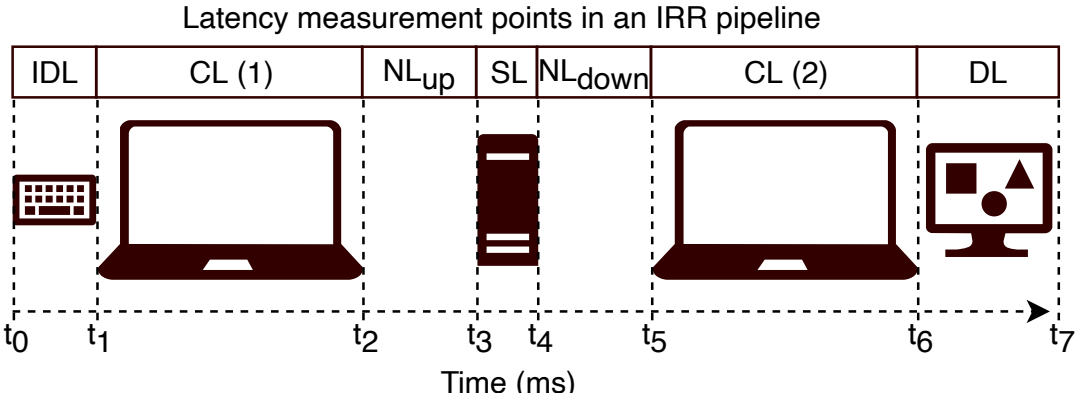


Figure 3.1 Interactive remote rendering latency model with measurement points. Interactions flow from left to right: input device to client to server, back to client and to the display. The sources of latency modelled are: Input Display Latency (IL), Client Latency (CL), Network Latency (NL), Server Latency (SL) and Display Latency (DL).

the input device (such as a keyboard). The interaction signal is processed by the client application where at t_2, its transfer across a network is started. At t_3, the server receives the interaction message from the client and begins to process it. After a period of time, the processed and rendered results from the server are transmitted across the network; this begins at t_4 and ends at t_5, where it arrives on the client device. The client software then processes (decodes, verifies, warps, etc.) the results received from the server and at t_6, initiates the display process and passes the pixels to the monitor. Finally, at t_7, the pixels are presented on the display to the user. From this information, we can compute the end-to-end by with $IL = t_7 - t_0$ or with:

$$IL = IDL + CL_1 + NL_{up} + SL + NL_{down} + CL_2 + DL \tag{3.1}$$

Interestingly, we can also calculate latency for various components. For example, we can determine SL by:

$$SL = IL - (IDL + CL + NL + DL) \tag{3.2}$$

or with the following if DL is not known:

$$SL = (t_5 - t_0 - .5 * NL) - (t_2 - t_0 + .5 * NL) \tag{3.3}$$

Manufacturers do not publish DL information and instead, only supply response times. Nevertheless, we can estimate the maximum DL from the monitor update frequency as a single frame time. NVIDIA's G-SYNC [69] technology aims to synch the GPU with the monitor refresh rate and will therefore significantly reduce DL. Nevertheless, the total IL, rather than the individual components of it, is typically the most important as this value is indicative of whether or not compensation techniques are required and what, if any, effects they have once implemented. It is therefore this end-to-end IL to which we refer for the rest of this paper.

3.3. Simulating Latency

Measuring and ensuring minimal IL is a critical task when building IRR systems. However, it is not practical to perform measurements over WAN as there is no control over the amount of latency introduced (making it difficult to repeat experiments), and measurements may be influenced by various factors that are outside of our control. Existing latency simulation tools (e.g. Clumsy [70] and TMnetSim Network Simulator [71]) yielded highly inaccurate results during testing and the majority of them cannot be integrated into our test applications, nor operate between two targeted applications. Therefore, in order provide a suitable and stable environment for measuring and testing IL, we built a latency simulator.

Simulating interaction latency is not as straightforward as inserting a delay either before or after an interaction has occurred, even though some authors appear to suggest that this is exactly what they did (for example [72, 5]). Instead, it is critical that interactions are delayed in an asynchronous manner. Figure 3.2 illustrates how asynchronous processing of interactions results in their inter-arrival times remaining constant. This is important because future interactions must not take longer to complete than previous ones, unless specifically designed to do so. If interactions are delayed synchronously, the amount of delay each interaction experiences will start to increase after the first interaction. However, this can only occur when the Send Delay (SD) (the delay between interactions) is less than IL. If $SD > IL$, synchronous processing may be used as a backlog of to-be-processed interactions will never occur. Consider the synchronous example of Figure 3.2.

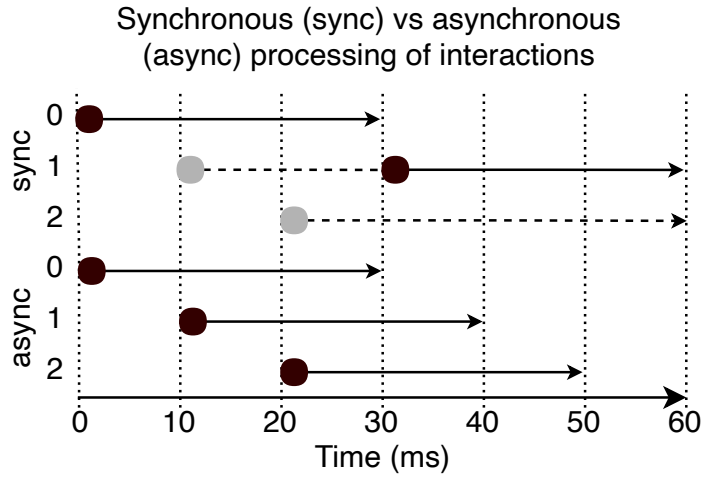


Figure 3.2 Synchronous vs asynchronous processing of interaction messages. Synchronous processing results in a backlog, while with asynchronous processing, the inter-arrival rate of interaction messages is constant and no backlog forms.

In the synchronous example, each interaction will be delayed sequentially, thereby adding an additional delay to future interactions and resulting in a backlog. This additional delay is represented by the dashed line in Figure 3.2. With each interaction talking longer than the previous (due to a growing backlog), we can calculate the expected delay for a given interaction (i) entering the latency simulator with:

$$I_{expectedDelay} = NL + n(NL - SD) \tag{3.4}$$

where n is the interaction number. Therefore, the delay experienced by each interaction will increase by $NL - SD$, per interaction.

The solution to this issue is to process the incoming messages asynchronously. Asynchronous programming is not easy to implement and can lead to concurrency issues such as interaction i_{n+1} being delayed and processed before interaction i_n. The reason for this is that there is no guarantee when a task will be started, only that it will be started.

To ensure that interactions will be processed, and their results delivered to the client (from the server) in the order in which they were created and sent, we built a latency simulator which delays messages and keeps track of how many have been received (M_r) and how many have been processed/released (M_p).

3.3.1. Implementation

The latency simulator has an input $Delay(message, duration)$, which delays a message for a specified duration, and an output $MessageReady(r)$, which is an event raised when a message has been delayed. *Delay* creates an object called LatencySimulatorResult, *r*, which has two properties: *message* and *messageNumber*. The incoming message is assigned to *r.message* and M_r to *r.messageNumber*. M_r is then incremented. If $duration = 0$, the $MessageReady(r)$ event is immediately fired signaling to any subscribers that a result is ready. If $duration > 0$, a thread is created and *r* is passed to it. When the thread starts, it immediately sleeps for the specified duration. Next, the thread waits until $M_p + 1 = r.messageNumber$. The $MessageReady(r)$ event is then raised and finally, M_p is incremented. The pseudo code for the latency simulator is as follows:

Algorithm 1 Latency simulation

```
1:  M_r = 0, M_p = 0
2:  procedure DELAY(message,duration)
3:      LatencySimulatorResult r
4:      r.message = message
5:      r.messageNumber = M_r
6:      if (duration == 0) then
7:          raise event MessageReady(r)
8:      else if (duration > 0) then RunThread(r, duration)
9:  procedure RUNTHREAD(r,duration)
10:     sleep(duration)
11:     while r.messageNumber! = M_p + 1 do sleep(1)
12:     raise event MessageReady(r)
13:     increment M_p
```

To test the latency simulator, we performed 5 experiments, each with a different latency value: 50ms, 100ms, 200ms, 300ms and 400ms. Later (see §3.7.1), we test this simulator in an IRR system. For each experiment, we collected 1000 measurements with a 100ms pause between each. Measurements were collected by using a .NET Stopwatch, timing the delays between adding a dummy message to the latency simulator and the point at which it exited – indicated with the triggering of an event. Table 3.1 presents the results of these experiments, and demonstrates that the simulator produces delays very similar to those intended.

Table 3.1 Latency simulator measurements

	50ms	**100ms**	**200ms**	**300ms**	**400ms**
mean	50.99	101.02	200.89	300.99	401.02
σ	0.72	0.73	0.70	0.65	0.69
σ^2	0.52	0.54	0.49	0.43	0.47

A very simple testing environment was then developed where a "client" and a "server" were emulated on separate threads. On the "server", rendering was simulated by sleeping a thread for 30ms. The latency simulator was integrated into this environment and repeated the above experiment 5 times, each time measuring the CL, NL and SL parameters as described in Figure 3.1. Table 3.2 shows the mean of each experiment performed with injected latencies of 50ms (other latencies produced similar results).

Table 3.2 Simulated model parameter measurements (ms)

CL_1	NL_{up}	SL	NL_{down}	CL_2	sum	IL
0.47	50.97	38.01	50.00	5.46	144.93	146.53
0.92	50.00	36.77	50.03	6.90	144.63	148.00
0.89	50.00	39.59	50.03	7.07	147.60	146.12
0.62	50.53	38.96	50.35	5.96	146.45	147.02
0.50	50.32	37.33	50.03	6.22	144.42	145.77

3.4. Measurement Approaches

From the literature, it can be said that there are two main approaches to measuring IL. We first describe these approaches in detail and briefly discuss another approach (§3.4.3) that we have identified. Although we were unable to find this approach is in the literature, it is straightforward enough and therefore assume that we are not the first to describe it.

3.4.1. Observer

Observer approaches involve writing software that "hooks" into the IRR system in order to take measurements. This is different from the integrated approach in that access to the source code is not required. No potentially expensive hardware is required and the process can be automated with simulated interactions.

Employing this approach, Chen et al. [34] attempt to measure the IL of a cloud gaming system, which consists of a client application and a remote server. To achieve this, they selected a game with a built-in menu screen which is activated by pressing the escape (ESC) key. To avoid the need for human input, the authors simulate the press of the ESC key with the expectation that after some period of time, a result from the server will arrive on the client and the client application will publish an on-screen update, thereby displaying the menu. When the ESC key is simulated, a time measurement (t_1) is taken. To measure the moment the response from the server arrives (t_5), the authors hook into the recvfrom() function, which is called when the client attempts to retrieve data from the socket. The authors then measure the time between data arriving on the client and the frame being presented on-screen. To do this, a library called Detours is used to hook into the underlying graphics API. Specifically, the authors intercept the IDirect3dDevice9::EndScene() function, which is a function from the Direct3D library that is called just before graphics are about to be presented on-screen, and measure t_6. For each frame that arrives on the client, the colors of a chosen set of pixels are inspected. When a change in pixels is detected, they conclude that the screen has been updated and then record the time (t_7). The difference in time between user input and identifying the change in pixels yields an IL measurement.

One potential issue with the approach taken by Chen et. al. is that their technique relies on there being an in-game menu. If the IRR system is not a games-based one, or if there is no screen update specific to a certain interaction, there is no way to determine whether or not the screen update is a result of an interaction or in-application mechanics. This issue may lead to incorrect measurements and potentially, make measurement impossible. Further, the authors did not compare their results with a system with known IL and therefore it is not clear how accurate or reasonable their results are. A more general disadvantage of this approach is that hooking will change the system behavior, even if in a small way. Hooking is not straightforward and in order to obtain their measurements, the authors had to perform complicated hooking techniques which may result in instability of the IRR system and also requires expert knowledge to perform. Finally, it is unclear how frames resulting from actual interactions (rather than background scenery updates) are identified.

3.4.2. Hardware

Hardware approaches are those that aim to measure IL using hardware and/or external devices. For example, Steed in [73], uses a high-speed camera positioned in front of a monitor on which simulated frames of a 3D object will be displayed. A pendulum is then set between the monitor and the camera. The pendulum, fitted with a light-emitting diode (LED), is tracked and its position information is sent to and used by a rendering application to “drive” the simulated frame. Using the camera, the operator video-records the scene and analyzes the resulting footage. IL is estimated by “counting the number of frames between pendulum and the simulated image”. While counting frames is a manual process, performed by the operator, and is

therefore prone to human error, this was later addressed by Friston and Steed [74], where they introduced an automatic frame counting algorithm.

An advantage of this approach is that there is potentially no impact on the IRR system, and provides full end-to-end latency measurements. However, the operator needs access to possibly expensive equipment (high-speed camera, tracker, LED, etc.), which needs to be set up and calibrated. Domain-specific knowledge may also be required to configure the hardware and the need for human intervention may lead to unreliable measurements.

3.4.3. Integrated

This approach involves writing IL measurement features alongside the IRR system, directly integrating measurement taking into system source-code. With the integrated approach, IL can be measured from t_1 to t_6 (see Figure 3.1), and is therefore not only capable of measuring end-to-end IL, but is also useful for debugging system bottle-necks as measurements can be taken at various points in the system, at the source of latency, giving more control to the developer over where and when IL is measured.

Measuring delays across a network will typically involve time synchronisation between all machines, and is challenging to perform [75]. However, when measuring end-to-end IL, this can be avoided by performing all measurements locally, on the client. To achieve this, when an interaction is registered on the client application of the IRR system, a GUID and stopwatch must be created, and the stopwatch started. The GUID and stopwatch must be stored in a data structure such as a dictionary.

The same GUID should be included in the interaction message sent to the rendering server and the result message from the server must include the GUID when sent back to the client. On the client, any arriving messages from the server should be inspected and the associated GUID can then be matched against the dictionary storing the stopwatches. The corresponding stopwatch should be extracted from the dictionary and stopped. IL can then be measured as the total number of milliseconds elapsed on the stopwatch. In order to measure at specific points, multiple stopwatches can be used and their results added to each message as it passes through the system components. The sum of these delays could then be subtracted from the total IL in order to calculate NL.

While this approach is robust and flexible, it does require access to source code, potentially making it unavailable to many. In addition, bandwidth usage will be increased (even just a little), and since the system will require modification, it will be affected by the implementation of this approach. There is also a development cost as well as the requirement to have expert knowledge of the IRR system and of technical programming.

Table 3.3 Interactive Remote Rendering measurement approaches: their advantages and disadvantages

Measurement Approach	
Observer	
Advantages	+ Can be automated with simulated interactions and automated logging. Might be cheaper than hardware approach as no expensive hardware is required. + Provides full event-to-result latency. measurements, including Display Latency (DL)
Disadvantages	- IRR system will be impacted, even if slightly (due to hooking the Graphics API). - Difficult to associate an action with a particular frame update. - Has measurement resolution equal to the display refresh rate. - Ignores Input Device Latency (IDL).
Hardware	
Advantages	+ Potentially no impact on the IRR system. + Provides full event-to-result latency, measurements, including IDL and DL.
Disadvantages	- Not easy to automate. - Requires human intervention, which may lead to unreliable measurements. - Requires hardware which may be expensive and difficult to set up. - Domain-specific knowledge is required to install, configure and use the needed hardware.
Integrated	
Advantages	+ Useful for debugging system bottle-necks as measurements can be taken at various points in the system. + Doesn't require complex OS input event stream hooking. + Provides accurate IL measurements if IDL and DL are not important. + No expensive hardware required. + Allows for more control and the ability to take measurements at specific system locations.
Disadvantages	- Requires access to source-code. - Will increase bandwidth usage, even if just a little. - Will require modification to IRR system and affect it in some way. - Ignores IDL and DL. - Technical programming knowledge is required.

Later, in §3.6, this method is implemented so that we can measure IL and use those measurements as a comparison with results from our new technique described in §3.5.
A summary of the advantages and disadvantages of the above approaches can be seen in 3.3. In the next section, we describe the simulation of IL, which we later integrate into an IRR platform purpose-built for exploring end-to-end latency and measurement approaches.

3.5. A Software-Based Interaction Latency Measurement Tool (LMT)

Existing approaches to measuring end-to-end IL are not generic enough for use on a wide variety of IRR systems, and indeed, typical rendering applications. As previously mentioned, it is a common requirement for source code to be accessible (e.g to tag messages), to hook into the system, etc., and the need for specialised system knowledge and possible overhead costs for hardware and configuration make measuring IL all the more challenging.

In this section we introduce our LMT, which avoids such issues.

To achieve this, we can monitor the images/frames generated by an application and identify the moment an interaction event occurs when two consecutive frames are significantly different. To calculate how different one frame is from another, we determine the PSNR. We built this approach into a simple tool, which is placed over the target application. The tool captures the screen as quickly as possible, time-stamping each image captured. Captured images are then analysed and the PSNR is calculated for consecutive images. In addition to this, interactions and the time that they occurred are recorded. When a significant drop in PSNR is detected, we can calculate IL as the difference in time between the moment an interaction was registered and the point at which a large drop in PSNR is detected.

To evaluate our tool, we developed a very simple Windows Form application. The background of this application changes color, from green to blue and back from blue to green. Color changes are triggered by keypress. The purpose of this application is to provide a test application, where there is no (or as close to zero as possible) IL.

An IRR simple system is then built and the integrated approach to measuring IL is built into the system, enabling us to measure its baseline IL. Afterwards, the LMT is used on the IRR system and results are presented.

3.5.1. Implementation

The proposed tool is a single Windows Form application which displays a red capture box (C_{box}) of 50x50 pixels, with a transparent background. The C_{box} must be placed over the display window of a target application (e.g. video game, video, etc.). A key, bound and used to initiate capturing of frames within the C_{box}, must then pressed. Captured frames are timestamped and stored in memory. During this period, the user must perform an action on the IRR system (e.g. press a key to move a character in a game, or "seek" through a video). The interaction, as well as the time of its occurrence, are both recorded. We trigger the capturing of frames separately so that the first frame associated with an interaction has a previous frame to compare with. Frame capturing results in a set of captures $C = c_1 ... c_n$ where c is a single capture of 50x50 pixels. After an interaction has been performed, the timestamps of the captured frames are searched for a time that corresponds with when the interaction occurred. When the corresponding frame is identified, IL is calculated as the difference in time between the timestamp of the frame and that of the interaction.

On start up of the LMT, we hook into the OS-level keyboard event stream and raise an event when an input is detected. A global stopwatch is also created and started. The C_{box} must be positioned over a target application which on keypress, causes some interaction to occur. Capturing is manually initiated and each capture is tagged with a timestamp from the global stopwatch.

When an interaction I is detected, the time of the interaction I_t is recorded from the global stopwatch. When measurement taking is complete (we terminate this via another key), the interaction times are matched with capture times. To do this, each image is compared with the previous on. In other words, C_n is compared with C_{n-1} on a per-pixel basis. If $C_{n[i,j]} \neq C_{n-1[i,j]}$, where $[i, j]$ represent pixel coordinates, we record IL:

$$IL = C_{n_t} - I_t \tag{3.5}$$

Where C_{n_t} is the n^{th} capture at time t and I_t is the interaction at t.

Since this approach directly compares two frames for differences between them, the C_{box} must be carefully positioned such that no pixel changes occur unless caused by an interaction. We can consider this environment to be "static" since frame updates are a direct consequence to an interaction. However, in typical environments (e.g. games, visualisations, etc.), frames may change due to the 3D engine mechanics (swaying grass, particles, Artificial Intelligence actions, etc.). This results in an issue where each capture is different to the previous and therefore determining which frame is a result of an interaction is impossible. To circumvent this problem, we calculate a PSNR for each captured image by comparing it with the one before it.

In a static environment where frames do not change unless caused by an interaction, C_n will have a PSNR of 100 when it is identical to C_{n-1}. In a non-static environment, every image will be different than the previous and therefore the PSNR will vary according to how similar the images are (no two images will be identical due to small scene variations). When an interaction occurs, the difference between frames should be more pronounced (unless large changes are occurring due to cinematics such, for example), and should therefore manifest as a dramatic and abrupt drop in PSNR, making detecting interaction-responsible frames possible. However, given that multiple frames can occur before an interaction-responsible frame is available, we need to find the first frame where a large change to PSNR is observable.

Therefore, for each capture, we calculate PSNR (equation 3.6) by considering each image and the one before it such that:

$$psnr(C_n) = \begin{cases} 100, & \text{if } n = 0 \\ PSNR(C_n, C_{n-1}), & \text{otherwise} \end{cases} \tag{3.6}$$

Where the *PSNR* is a function which returns a *psnr* calculated with:

$$PSNR(C_n, C_{n-1}) = \begin{cases} 100, & \text{if } \varepsilon = 0 \\ 20 * log_{10} * \frac{255}{\sqrt{\varepsilon}}, & \text{otherwise} \end{cases} \tag{3.7}$$

and ε is the Mean Squared Error between captures C_n and C_{n-1}.

Using the timestamp associated with the capture, IL can be measured as the difference in time between an interaction timestamp and the timestamp associated with the first detected drop in PSNR for a given capture:

$$IL_n = C_t - I_t, when |C_{psnr} - C_{psnr-1}| > \theta \tag{3.8}$$

Above, I_t is the timestamp of the interaction and C_t is the timestamp of the of the first capture which drops below some threshold θ, which we set as the mean PSNR at rest, without interaction; this captures the "ambient" noise between captures.

Figure 3.3 shows the LMT positioned over a simple, and separate, Windows Form application built to test the maximum rate at which frames can be captured. This works by pressing a key, which causes the background of the Windows Form application to change color from green to blue. Since there are no background or rendering processes, the color change should be near-immediate and therefore have as close to zero IL as possible. We used this tool to measure the baseline of our LMT, finding that it can detect latency as low as the refresh rate of our monitor, which is ≈ 16ms (60Hz).

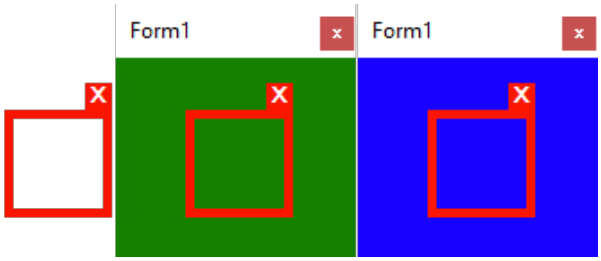


Figure 3.3 A tool built to measure the baseline performance of our latency measurement tool. The C_{box} (red bounding box) is used to select pixels for monitoring. The C_{box} is placed over the green window (center) which changes to blue (right) when an action is performed. Interaction Latency is measured as the delay between performing the action and the window changing from green to blue.

3.6. Experimental setup

In this section we describe an IRR system built to provide a platform from which IL measurements can be collected using the introduced LMT in §3.5. In addition, the IRR platform affords us three benefits: i) Latency simulation can be tested more vigorously; ii) IL measurements can be collected in both remote (rendering performed remotely) and local-only modes (all components on the same machine); iii) The LMT can be evaluated on both a local-only and remote rendering application. The aim of the experiments is to validate the LMT and to demonstrate that it is able to reasonably measure IL.

All development and testing (including that of the LMT) was conducted on a 15" MacBook Pro 2017 with a dedicated Radeon Pro 560 (4GB DDR5 RAM) graphics card, running Windows 10 in Bootcamp mode.

The IRR system is composed of a simple client/server architecture and has the latency simulator integrated into it. In order to ameliorate cross-platform compatibility issues and to allow us more control, we used Unity3D for both the client and server applications. Further, this allows us to leverage not just the rapid prototyping capabilities of Unity3D, but also those of certain features such as data types and compression/decompression utilities. Communications between the client and server applications are performed using TCP[1]. We measure the end-to-end IL of the IRR

[1]While UDP would provide lower latencies, complications such as packets being out of order or being dropped, may occur. TCP was therefore chosen for its reliability at the expense of delay.

platform using the integrated approach described in §3.4.3 so that we can compare the collected measurements with those of the LMT.

3.6.1. The client application

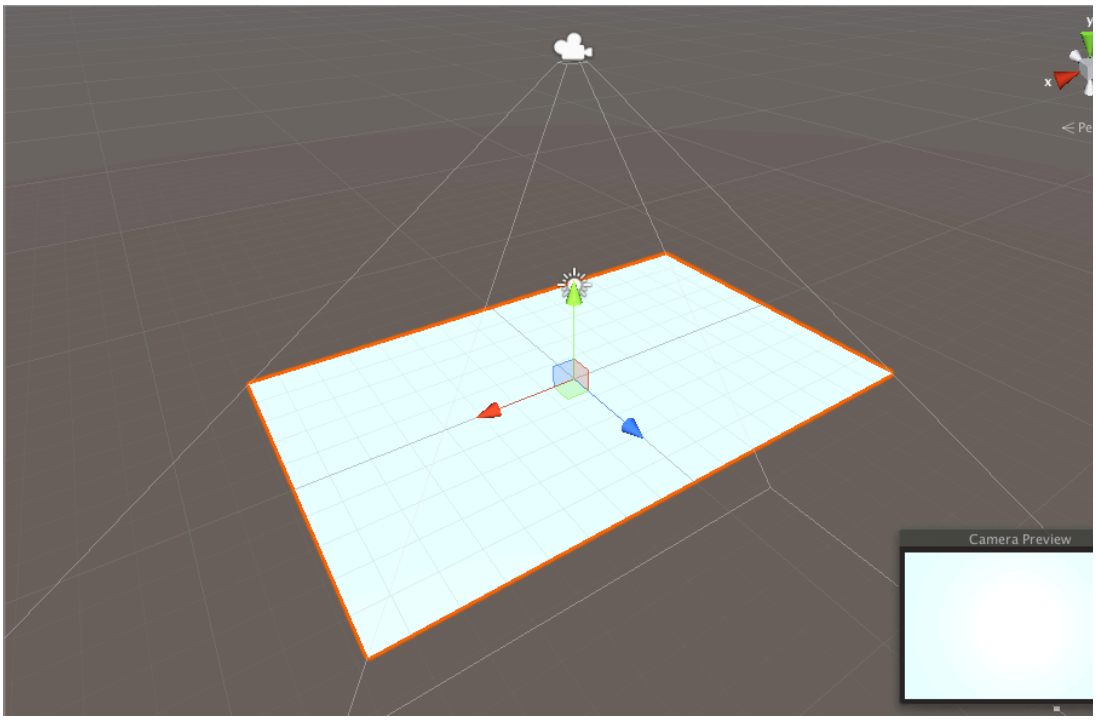


Figure 3.4 The scene view of the client application. The frame data received from the server is converted to a texture and applied to the plane, which the user observes through the camera.

The client application consists of a simple scene with a camera and a plane (Figure 3.4). The plane is used to display the images from the server, which are mapped onto the plane before being rendered by the camera. Interactions are fed to the system via a template which consist of either an “a” or a “d” per line. The interactions are loaded into a queue when the application starts up and messages are then sent to the server application at a rate of 10 messages per second. Messages arriving on the client (from the server application) are first fed through the latency simulator, delaying arriving messages for a specific amount of time.

3.6.2. The server application

The server application has a scene consisting of a camera and a 3D cube. When a message arrives from the client, it is put into the latency simulator and delayed for a specific amount of time. Next, it is deserialized and an event is raised, signaling that the scene must be updated. To update the scene, the interaction of the arrived message is inspected. If the interaction is an “a”, the cube is rotated left. If the interaction is a “d”, the cube is rotated right. Once a transformation to the cube is complete, the render result is captured (the game view, shown in Figure 3.5) and compressed to JPG at 75% compression ratio, which is the default value supplied by Unity3D. The message is then serialized and sent back to the client. While the rendering scene is simple

(just a rotating cube), it is sufficient – the primary purpose is to perform rendering and thus add a variable rendering delay to system, as well as to produce render results/frames.

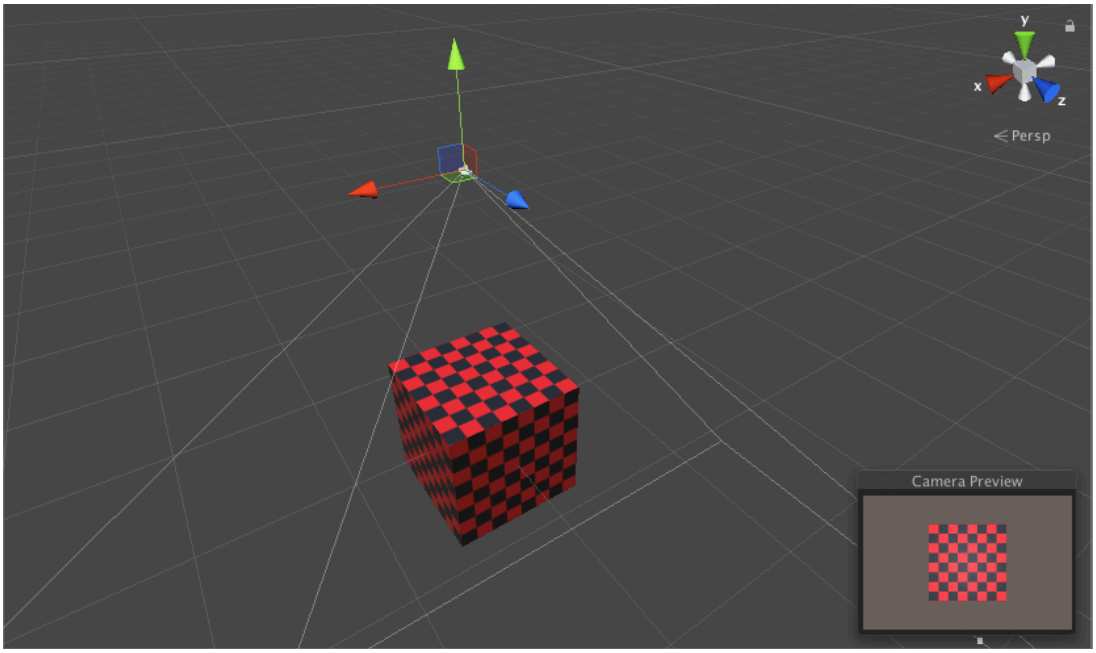


Figure 3.5 A scene view of the server application shows a 3D cube with a camera positioned above it.

3.7. Experiments and Results

In this section, we present a number of experiments and their results. First, we measure the base operating IL of the IRR platform, as well as when used with simulated latencies of 50ms and 100ms. We do this to get an understanding of the expected amount of latency to be measured, and to be able to compare those results with ones collected using the LMT. We also perform measurements of IL over WAN and compare the results with a similar – but simulated – amount of latency. This allows us to validate that the system is able to operate correctly in an environment where NL is non-simulated. We then perform measurements with the LMT before comparing the results of the LMT with those of the IRR system. Note that all measurements of the IRR system were captured using the integrated approach (see §3.4.3).

3.7.1. IRR Measurements

We measured the base latency (without simulated NL) of the IRR system by taking measurements at t_1 and t_6 (as described in Figure 3.1). Here, the client and server applications were run on our local system (located in Cambridge, UK) by feeding into it 1000 predefined actions at an arbitrary interaction rate (SD) of 10 interaction per second (or 1 interaction every 100ms). The mean measured base IL (IL_base) was measured to be 15.19ms (Figure 3.6). After establishing a baseline latency for the IRR system, a further 3 (again, locally) experiments were performed on our local machine, each time injecting a different amount of latency into the system: 50ms, 100ms and 174ms (this value was measured between Cambridge and Northern

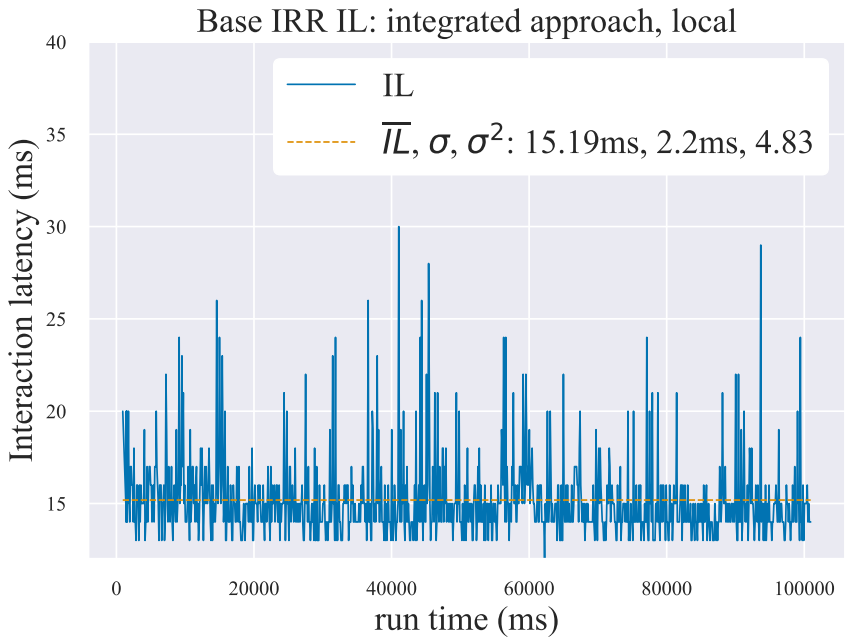


Figure 3.6 Base interaction latency measured with the integrated approach. Simulation was run on a local machine.

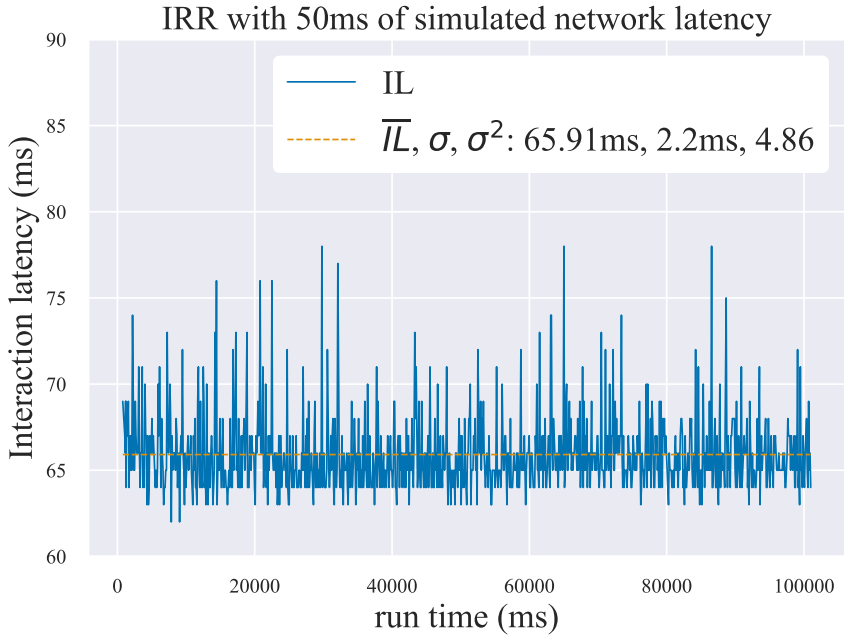


Figure 3.7 Interaction latency measured using the integrated approach with an additional 50ms simulated latency.

California, using the Ping utility). Our expectation was for the results to show that $\overline{IL} = NL + IL_{base}$

In both Figure 3.7 and Figure 3.8 it can be noticed that the mean IL measured is $\approx$ 16ms more than the simulated *NL*, which is a further indication that the latency simulator is performing as expected since our measured base latency is $\approx$ 15ms. The IRR system was then run over WAN, with the server application hosted remotely on an Amazon EC2 instance located in Northern California. We did this three times: one early morning, one in the afternoon and one at night.

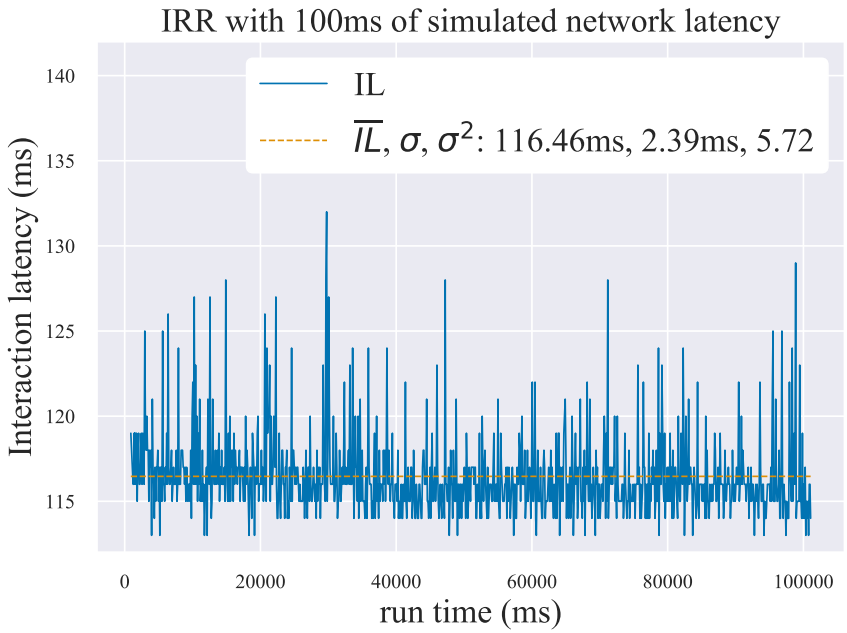


Figure 3.8 Interaction latency measured using the integrated approach with an additional 100ms simulated latency.

We did this to counter variation in broadband speeds commonly experienced as "throttling", most often encountered during peak traffic hours. The mean IL measured during these experiments was 191.72ms, which we compare with the previous experiment where we introduced 174ms of simulated latency.

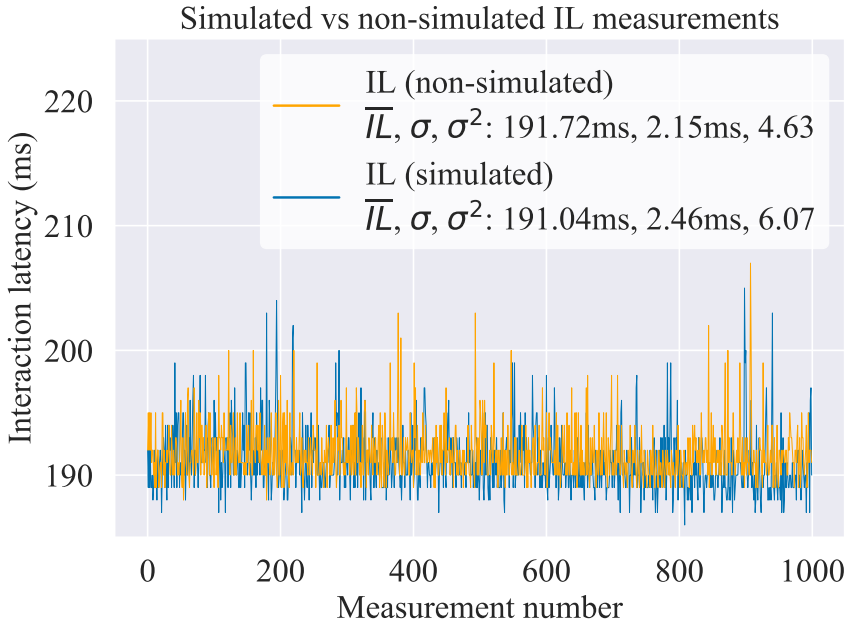


Figure 3.9 Simulated vs non-simulated (WAN) Interaction latency measured using the integrated approach. Zoomed-in view as signals are nearly identical and overlap otherwise.

Figures 3.9 and 3.10 show that the latency simulator, combined withe the IRR system, produces a delay similar to that of a real-world network, with the difference in signals being, on average, less than 1 millisecond. This is a clear indication that the latency simulator performs as expected,

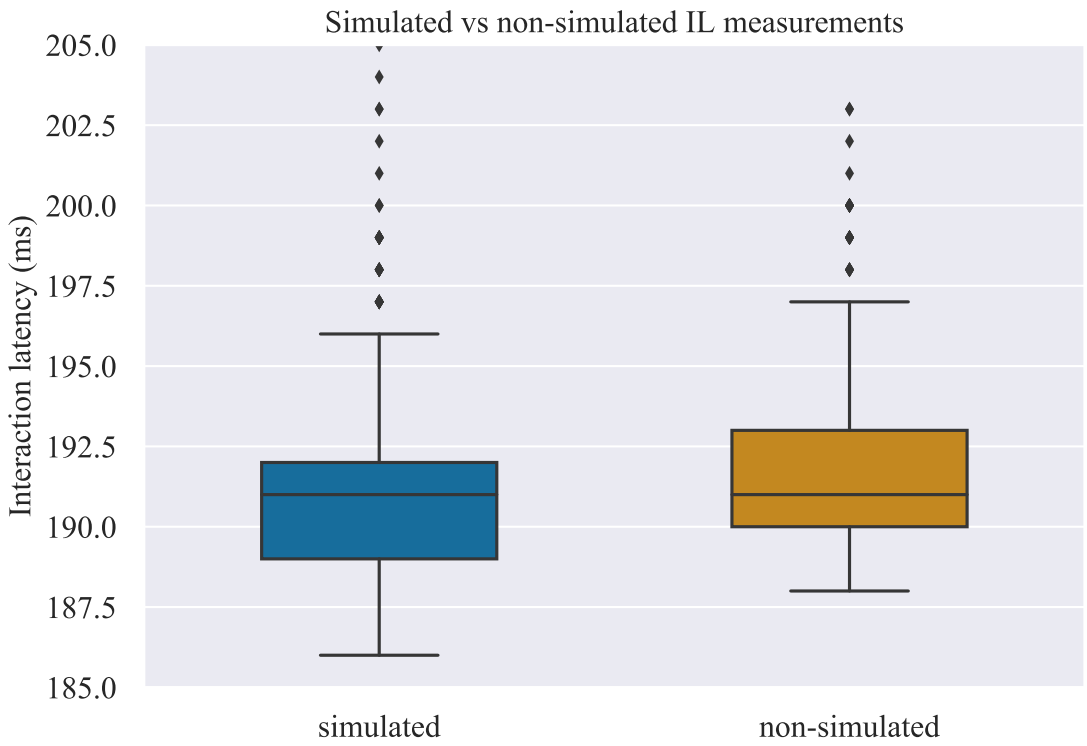


Figure 3.10 Boxplots showing means of simulated vs non-simulated Interaction Latency measurements.

which is further illustrated by Figure 3.11 and Figure 3.12, where it can be seen that both sets of measurements come from the same distribution. Note that there is no emulated jitter: the variation in the simulated latency is a result of threading and the testing on a non-realtime Operating System (OS) (Windows).

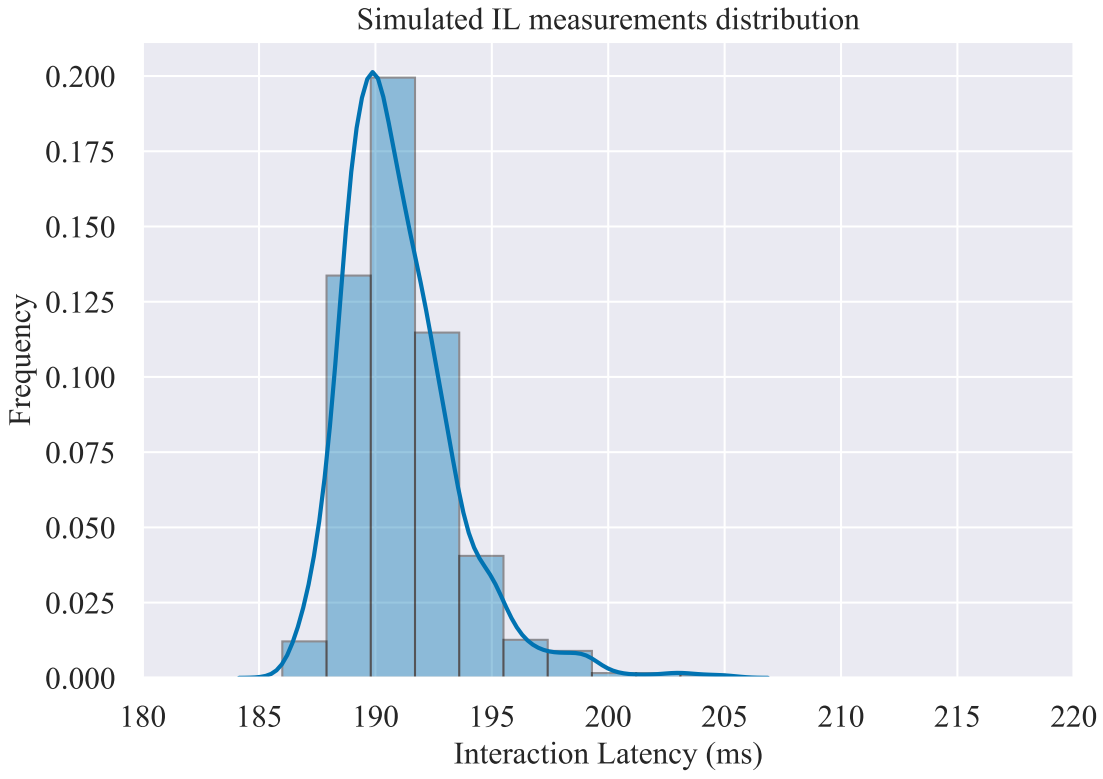


Figure 3.11 Distribution of Interaction Latency measurements collected when latency is simulated.

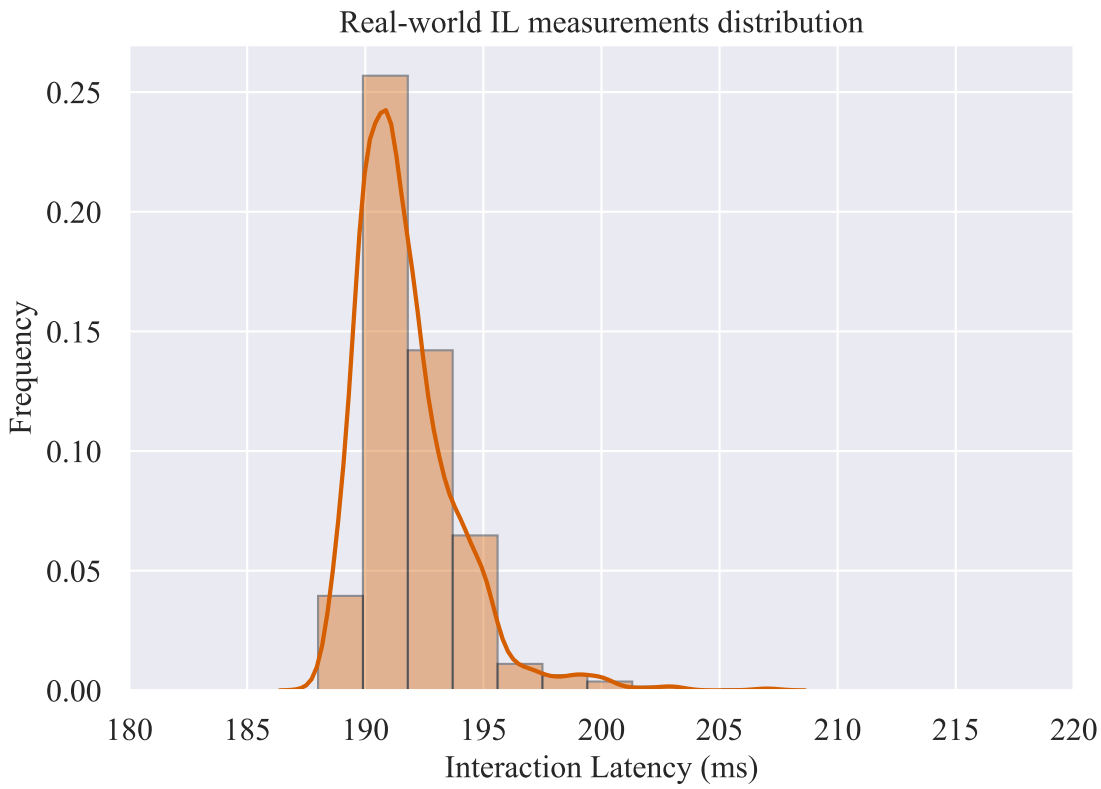


Figure 3.12 Distribution of Interaction Latency measurements collected when latency is produced by a real-world network.

3.7.2. LMT Testing and Evaluation

In order to determine the lowest amount of latency measurable (resolution) by the LMT, we built a simple Windows Form application, which we described in §3.5.1 (see Figure 3.3). We built the application so that we can test the baseline latency of the application without additional delays such as rendering time or network latency. There were no other tasks or applications in operation during experimentation and the color change should therefore have been immediate.

On a local machine, C_{box} (the red bounding box) of the LMT was placed over the green background of the test application and performed 50 manual key presses, with the expectation that our measurement results should indicate that $IL \approx \overline{C_t}$ (our display has a refresh rate of 60Hz).

Our results show that the LMT measured an average IL of 15.92ms, as can be seen in Figure 3.13, which is very close to the expected $\overline{C_t}$ of 16ms (+- 1ms). The discrepancy of 1ms is due fluctuations in the timing of thread-starts and stops. We then repeated the experiment and collected 10 new measurements but used the IRR system (described in §3.6) rather than the test application.

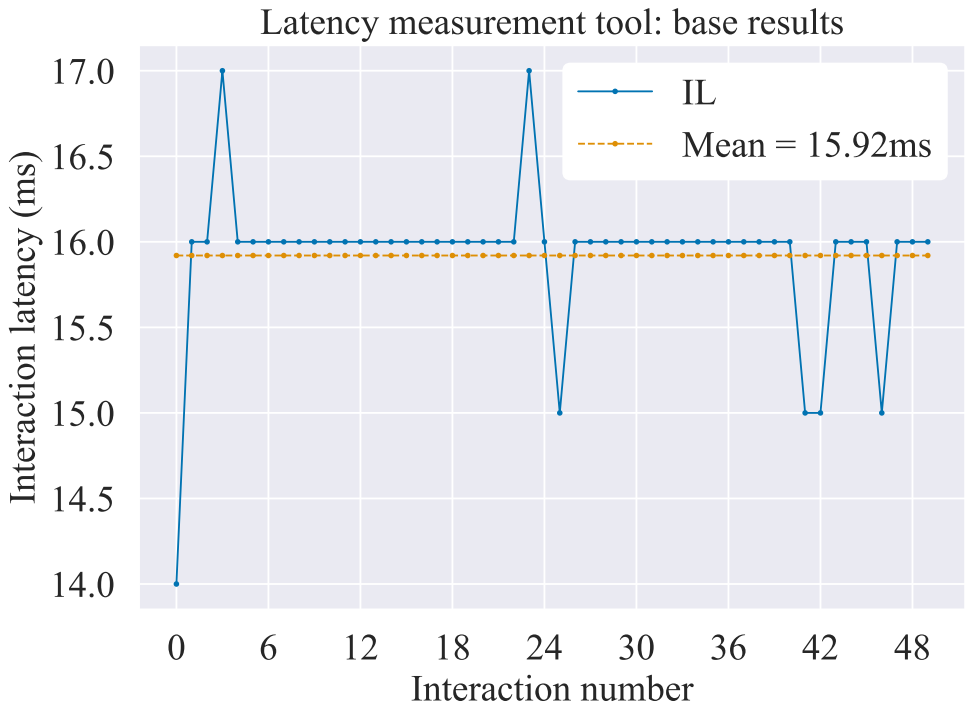


Figure 3.13 Latency measurement tool base results collected using a test-application which simply changes a Windows Form background color from green to blue, when a key is pressed.

During the analysis of the results obtained with the LMT, we found clear visual indications that PSNR does indeed drop enough to enable the positive identification of frames resulting from interaction, and therefore the calculation of IL. For instance, Figure 3.14 shows pronounced drops in PSNR values for the 10 interactions collected over 20 seconds, dropping dramatically from 100 to $\approx$ 27 when a frame is significantly different from the one before it (due to interaction).

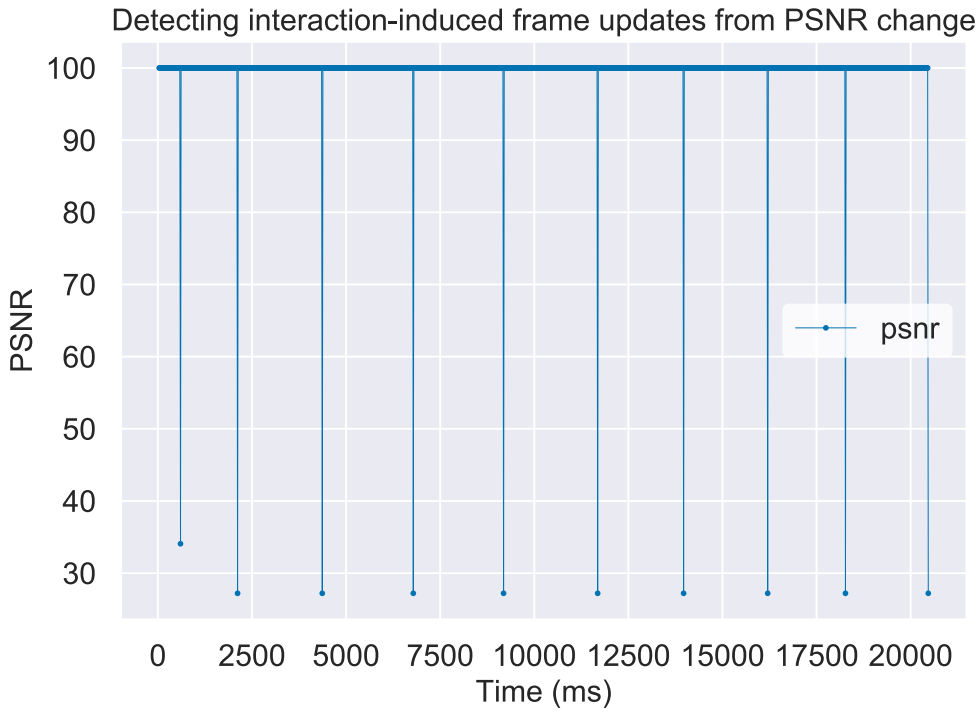


Figure 3.14 10 interactions performed over 20 seconds. Drops in PSNR shows that a significant difference between consecutive frames has been registered, indicating that an interaction has occurred.

When zooming into the data of Figure 3.14, it can be seen that there is a visible delay between when an interaction is performed and when the on-screen image is updated. For instance, in Figure 3.15, an interaction is performed at t_1, which occurs just before a frame is captured. At t_2, which in this case is the next frame, we detect a large drop in PSNR. Using the timestamp of the image associated with t_2, we can calculate $IL = t_2 - t_1$.

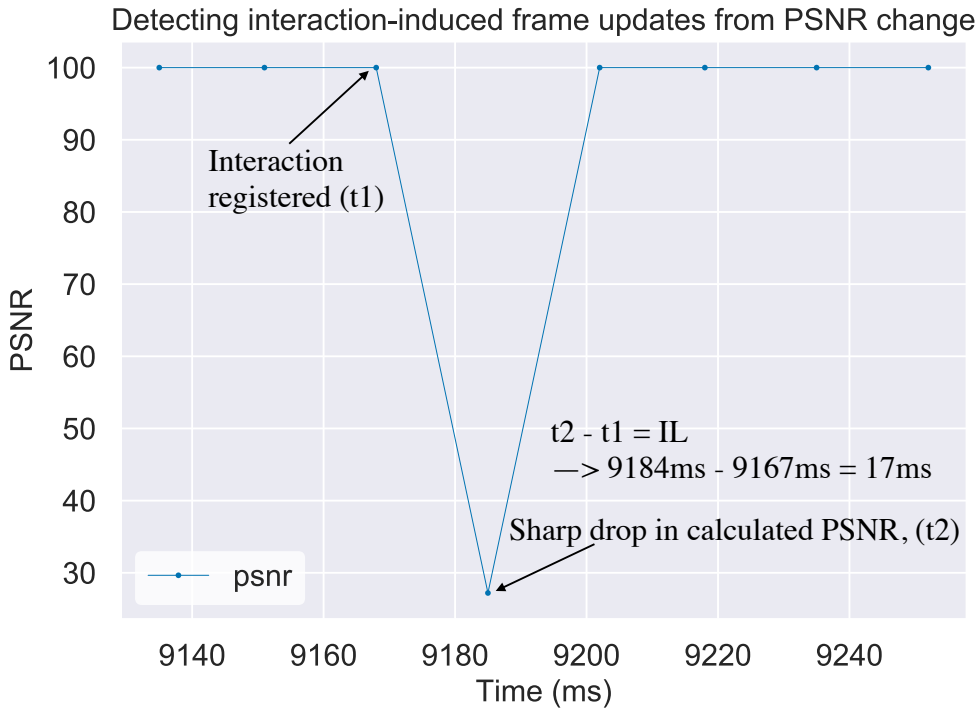
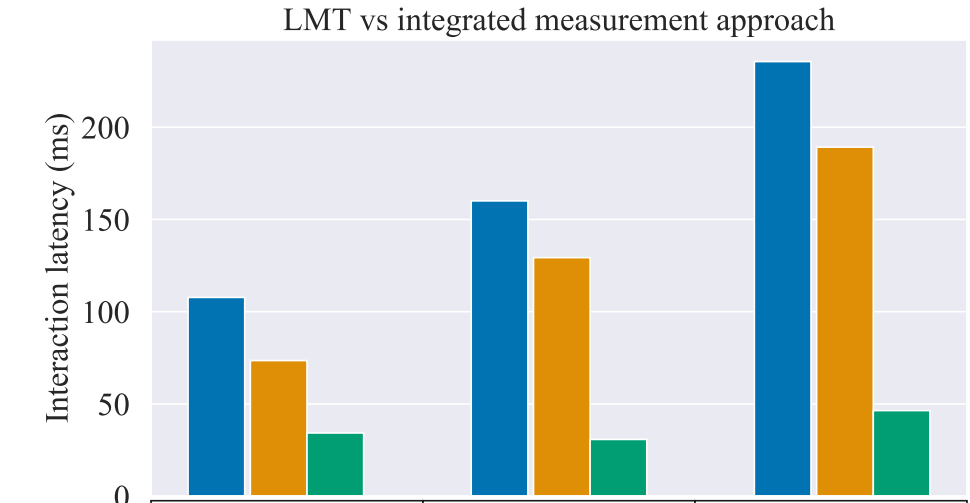


Figure 3.15 Peak Signal to Noise Ratio (PSNR) used to identify change in scene from interaction. An interaction is performed and 17ms later, a drop in PSNR is detected.

Having validated the expected operation of the LMT, it was then tested on the IRR system with simulated latencies of 50ms, 100ms and 174ms. Since the LMT does not simulate interactions (and therefore requires manual input), only 10 measurements per experiment were collected. During these experiments, we also collected measurements using the integrated approach so that we can compare them with those obtained using the LMT. Figure 3.16 shows these results. Both the integrated and LMT measurement pairs were collected from the same experiment, however due to the measurements being collected via different applications, it is not possible to present the results along a unified timeline. From the figure, the LMT is shown to capture more latency than the integrated approach, and the delta (difference in IL measured between the LMT and integrated approach) remains relatively consistent across all latencies, whether simulated or not.



IL (ms)	50ms	100ms	WAN 174ms
LMT	107.74	160.05	235.58
Integrated	73.5	129.3	189.2
Delta	34.24	30.75	46.38

Figure 3.16 A comparison of measurements between Latency Measurement Tool and integrated approach. Measurements reported are averages of 10 measurements per experiment, repeated 3 times each. Delta is the difference between the two measurement approaches.

In all, the LMT has been demonstrated to be able to measure IL reasonably well. In the next section, we present a discussion.

3.8. Discussion

We modelled and developed a method for simulating latency and integrated it into a purpose-built IRR system so that we could test and evaluate a novel method for measuring the end-to-end IL of IRR systems. This section discusses our work from a high-level, indicates some of the potential issues with our work and proposes future work.

3.8.1. On simulating latency

In order to test latency measurement techniques, we needed a controlled environment with the ability to "inject" arbitrary amounts of latency. The tools available for simulating latency were

found to be inconsistent and unreliable during testing, and being a standalone application, unable to be integrated directly into our IRR system. Therefore, a purpose-built latency simulator was developed (see §3.2).

We measured our latency simulator and are confident that it behaves reasonably well when compared with our WAN experiments. In all experiments measured using the integrated approach, the mean IL was $\approx$ 15ms to 18ms above the introduced latency (see Figures 3.7 and 3.8). This is the expected result as our IRR system has a similar mean baseline (Figure 3.6). It is clear, though, that there is noise in our results. We do not simulate any jitter, dropped or otherwise Out-of-Order (OoO) packets. Therefore, we suspect this variability is due to the 15ms time-slice limitation of the Windows OS (described in §3.8.2), and the fact that the OS does not guarantee *when* a thread will begin, only that it *will*. In going forward, options to control jitter, OoO and the dropping of network packets would lend well to testing the robustness of the LMT tool and the system as a whole. We suspect that only the OoO packets will present an issue for the LMT, resulting in the incorrect frames being detected as "corresponding to an interaction".

3.8.2. On the IRR system

While simple, the IRR system described performs real rendering (and therefore introduces the variable rendering delay). We do not believe that a more complex scene (e.g. a video game) is required to test latency measurement approaches.

We built the integrated IL measurement technique into the IRR system to (i) measure IL as "close to the source" as possible, as well as to provide a baseline measurement of the system from which we could test and evaluate our introduced end-to-end IL measurement software tool. During measurement collecting, it was found that in addition to the noise generated by latency simulation, the IRR system introduces additional noise. For example, in Figure 3.6 and others, there is a large amount of variability in the data. While rendering latency is not consistent and will therefore add variability to the measurement, there may be an underlying process which is causing instability in our IRR system. If there is, the most likely cause is our chosen method for "pausing" a thread (we use the standard .NET Thread.Sleep() function). Windows-based machines are not real-time OS and provide a thread time-slice of 15ms, meaning that the sleep function has a resolution of 15ms, too. It has been noted that the time-slice can be configured to 1ms by using *timeBeginPeriod* and *timeEndPeriod*, however, we were reluctant to do this as it would affect the thread time-slice OS-wide and may have unexpected consequences. While we believe that this system noise is of little concern, we would like to reproduce and further verify our approach with a real-time OS.

3.8.3. On the Latency Measurement Tool

Our aim in developing the software-based LMT was to enable researchers and developers to measure IL without any hardware or configuration, while limiting the impact of measurement

taking on the IRR system. Latency measurement can be challenging to perform, especially if the application source code is inaccessible. The LMT expands on and improves the observer approach (such as in the example of Chen; described in §3.4.1), which is useful, but has a number of issues such as being unable to consider measurements of applications consisting of moving entities which are not coupled to user interaction (e.g. grass blowing in the wind) and the requirement to have expert coding knowledge. We expanded on this approach by capturing screen pixels within a user-controlled C_{box} and by monitoring a subset of frame pixels and using the PNSR between images to identify a frame resulting from interaction.

This approach means that the IL of potentially any rendering system can be measured, without any setup, expensive equipment, calibration, access to source code or expert knowledge. However, it is important to note that this approach will only capture end-to-end latency and therefore is unsuitable for scenarios in which specific parts of an IRR system need to be measured. Another limitation is that the LMT has a mean capture time $\overline{C_t}$ dictated by the monitor refresh rate (60Hz), which in our case was $\approx 16ms$. There is therefore a risk that a measurement will be out by at least $\overline{C_t}$ (if pixel change occurs before or after the capture). The bounds in which pixels were captured was arbitrarily set to be 50x50 pixels in size, which raises the question: would a smaller bounds (e.g. 10x10 pixels) result in similar measurements and how would this impact the performance and/or resolution of the LMT? Further, we did not test our LMT on a scene with moving background entities, however we believe that doing so would not impact the validity of our results: the drop in PSNR would be less pronounced, but the drop would still be evident - unless drastic scene changes occur such as an explosion in a video game. We therefore leave such an experiment to future work, too.

At present, performing measurements with the LMT is a very manual and slow process, requiring the operator to physically initiate capturing and then perform interactions. However, it should be possible to automate this process by simulating key presses.

It should be noted that IDL and DL were not measured during our experiments: comparing measurements from our LMT (and indeed the integrated approach) with those captured when using the hardware approach would help to further evaluate both techniques. Finally, multiple inputs can contribute to a single frame. Since we hook into the underlying OS event stream, we capture all inputs and for each, consider the first detected image response. In this way, all previous inputs are considered in our measurement, regardless of how many were used in generating a given frame.

3.8.4. Summary

The ability to easily and accurately measure IL is important but current methods are complex, often requiring expert knowledge, calibration, potentially expensive equipment and in some cases, access to source code. Towards this, our contribution is two-fold.

A method for simulating interaction latency

Being able to simulate IL with controllable latency parameters is important for assisting researchers and developers during the design and implementation phase of IRR systems, as well as when investigating potential latency compensation techniques. This research has therefore introduced, tested and validated a method for simulating latency. It was found simulating latency in IRR systems is a non-straightforward process and therefore a technique which provides fine-grained control over latency within less than 1ms of a real-world signal (see 3.9) was developed, with measurement results appearing to come from similar (normal) distributions 3.11. Additionally, it was found that asynchronous processing of interactions within the latency simulator is a critical factor. To validate the introduced latency simulation approach, multiple experiments were conducted using simulator-generated latencies and their results were compared with real-world latency measurements.

A novel framework for measuring interaction latency using a software-based observer approach

Three latency measurement technique categories were introduced: integrated, observer, hardware. Additionally, it was described in detail how to apply the integrated approach for measuring IL, which to the best of our knowledge has not been presented in literature. During measurement experimentation, it was found that existing approaches are sufficient when access to source code is available. However, all approaches lack generality and are often complicated to configure and use, limiting applicability. Therefore, these concerns were addressed by developing a novel general-purpose software-based method (LMT) for measuring end-to-end IL. In fact, LMT appears to be suitable for measuring the IL of any display system (local or remote) and can even be used on web applications such as YouTube and Netflix.

Chapter 4. N-Grams for predicting keyboard-based user interactions

4.1. Overview

The aim of this chapter is to introduce and explore N-Grams as a means for predicting keyboard-based user interactions. After briefly mentioning assumptions and constraints, the experimental setup is discussed. Next, the model is introduced, evaluated for accuracy and two implementations are explored and measured, followed by the presentation of results. Later, in §5 this prediction method is integrated into a simulator test bed and then into a real-world IRR system. Finally, a summary is provided.

4.2. N-Grams and their suitability for IRR systems

N-Grams are statistical models used in computational linguistics and probability, and are well known for their use in predictive text [76, 77] and sentiment analysis [78, 79]. Essentially, N-grams are adjacent n-tuples (gram) of words or characters which collectively describe the frequency of a "gram" (pattern) within a text or speech corpus, the results of which may be used to derive probabilities of observing a pattern, given some history. They are well suited to many tasks, including player goal prediction [80] and user actions modelling [81] in computer games. According to Rabin [82] N-Grams are powerful & computationally inexpensive, which provides the ability to model user actions with high accuracy. In fighting games such as Tekken, Mortal Kombat and Street Fighter, N-Grams are used [83] by Artificial Intelligence (AI) opponents to model user actions and offer counter moves. So successful are N-Grams at predicting which move the player will perform next, that error must be introduced to the model so that players have a chance at winning. N-grams may be used in either offline mode, where the training data is static, or in online mode, where the training data changes over time and therefore allows the model to adapt to changing patterns in user behavior during execution- time. The low computational requirements, accuracy and speed of N-Gram statistical models should make them well suited to predicting future user interactions, without significantly impacting IRR systems. The use of N-Grams for user interaction prediction in IRR systems does not appear to have been investigated in literature. For this reason and the benefits listed above, N-Grams were chosen as the primary method for predicting keyboard-based user interactions.

4.3. Assumptions and Constraints

4.3.1. Keyboard-only interaction prediction

While some work has been done on predicting mouse movements [84], albeit not specifically for interactive visualizations, keyboard interactions were chosen as the focus of this research due to time constraints and the added complexity of having to design two models, rather than one. Additionally, a number of issues were observed with mouse simulation (discussed later), contributing to this decision.

4.3.2. Interaction types

In a Virtual Environment (VE) such as an open-world games, city simulation or urban visualization (e.g. Google Street View), users can navigate using the keyboard. For example, by pressing the keys W, A, S or D, the player/camera can be moved forwards, left, backwards or right. Similarly, other keys, or combinations of keys, may be used: a "W, A" could signify a "forward-left" movement, or the user might rebind interactions to any of the standard keys at their disposal. Therefore, the N-Gram model must be able to cater for unknown keyboard interactions and their possible combinations.

4.4. Definitions

This chapter introduces the idea of predicting user interactions. User interaction prediction is later used to mitigate IL and as such, it is worth briefly describing some of the notation used in this chapter as the same notation will be used later on.

Prefix: This represents the interaction "history" used when making a prediction. In our case, a prefix is an array of the last N interactions or keys pressed.

N-Gram Order or just "Order": This is the length of the history used.
N: In this chapter and throughout the thesis, N is used to denote an integer. For example, we might say that an N-Gram Order has a length $N = 5$.

Corpus: A corpus is a collection of words or letters. In this thesis, we use individual letters to represent interactions.
C: This represents the number of times ("count") the prefix is observed within the corpus.

4.5. Experimental setup

In order to build, train and test an N-Gram model for keyboard-based user interaction prediction, data is required. Unfortunately, no user interaction databases appear to be available online and so there is no readily available source of data on which an N-Gram model can be trained and tested. Therefore, a simple application from which interactions could be collected was built.

While a user will likely only be aware of pressing the key and then releasing it, the operating system (OS) will repeatedly raise the “KeyDown” event while the key is held down; this will happen at a rate of about 30 times a second [36]. This stream of interactions will be used as the data for model training and testing and should make it possible to infer likely future interactions.

Using the Unity3D game engine and a free tool (Maze Generator) available on Unity’s Asset Store, 5 randomly generated VEmazes were constructed. In order to prevent random wandering and make interactions more purpose-like, ten collectable items were put into each generated maze at random locations. A sphere, representing the player, is controlled and allows the user to navigate the environment with the forward (W), left (A), backward (S) and right (D) keys. When the sphere collides with a collectable item, it is removed from the maze. Each maze was played and the first one thousand interactions from each attempt were recorded. Only keyboard interactions were recorded for use as an input data source to the N-Gram learning model. All five thousand interactions were then merged into one dataset. Finally, the dataset is divided into a training and test set.

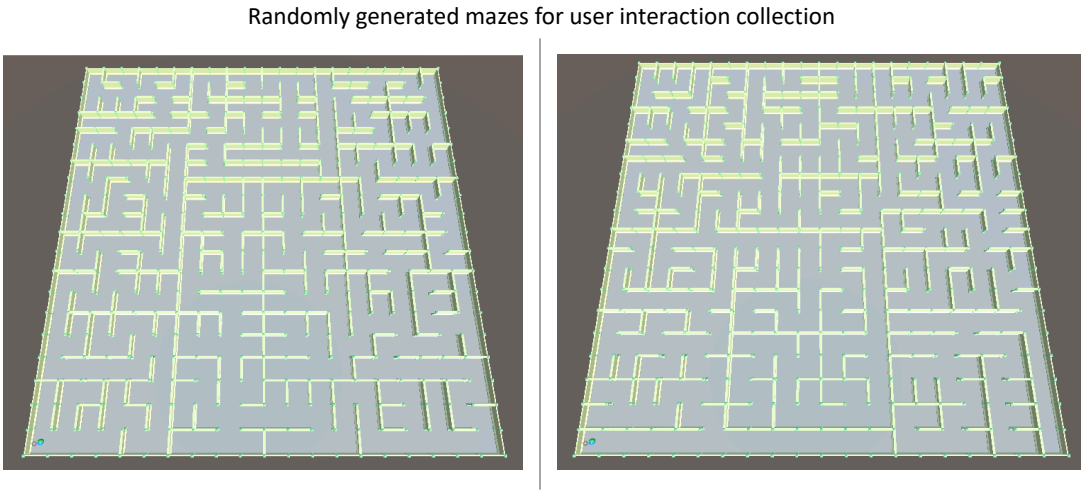


Figure 4.1 Randomly generated mazes used for gathering interaction data.

4.5.1. Limitations

Since there is no publically available data set of user interactions, one had to be created manually. This means that the data collected and used to test and train the N-Gram model may not be representative of the interaction style of the population at large. Additionally, the number of interactions allowed has been restricted to 4 (W, A, S and D), and no mouse input is modelled. Nevertheless, these limitations do not invalidate the performed experiments or their results. This is because N-Gram models adapt quickly to new input and their learning rate can be controlled by adjusting the amount of history used for training.

Table 4.1 Example corpus of user interaction events

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Event	A	A	A	W	W	W	D	D	A	S	S	S	S	W	W	A

4.6. The model

The primary data off which the N-Gram model works is a set of events. This set of events is known as a "corpus" where each event is either performed by a user or simulated. For the purposes of predicting user keyboard interactions, they can be stored as a FIFO (First-In-First-Out) queue of keypresses such as "w", "a", etc. These events can be referred to as "interactions" or "interaction events".
The immediate last n interactions of the corpus are known as a "prefix" and the length of the prefix is known as the order.

$$prefix = I_{n-N+1}^{n-1} \tag{4.1}$$

where I_n an interaction or element within the corpus, N is the order and n is the total number of elements within the corpus. The first step in predicting with N-Grams is to scan the corpus and count the number of times the prefix occurs. For each prefix match, the immediate next interaction is inspected and a count of how many times a particular interaction event follows a given prefix, is maintained. The interaction which occurs most frequently following the matched prefix is the most probable interaction to occur next. In other words, given a set of interactions, predict the next interaction (I_{i+1}) given some history and a frequency distribution:

$$I_{i+1} = P(I_i | I_{i-1}, ..., I_{i-N}) \tag{4.2}$$

As an example, if a user has been interacting with the system, the corpus has grown to consists of 10 interactions and we consider just the last two interactions ($N = 2$) for predicting the next:

Number of elements in corpus $n = 10$
$N = 2$
$Corpus =$ "W,A'',"W'',"W'',"A'',"W'',"W'',"A'',"S'',"W'',"W''
$Prefix =$ "W'',"W''

From this, bigrams can be constructed, which are tuples of adjacent elements. For example, the resulting set of bigrams (Since N=2), excluding the prefix, from the above would be:

$bigrams =$
$\{(\text{"}W,A'',\text{"}W''), (\text{"}W'',\text{"}W''), (\text{"}W'',\text{"}A''), (\text{"}A'',\text{"}W''), (\text{"}W'',\text{"}W''), (\text{"}W'',\text{"}A''), (\text{"}A'',\text{"}S''), (\text{"}S'',\text{"}W'')\}$

From the above, it can be seen that the number of times the prefix “W”, “W” occurs is two. The number of times each interaction has occurred immediately following the prefix is then counted. Since only interactions of type “W”, “A”, “S” and “D” have been observed, then:

$$\text{“}W\text{”}:0, \text{“}A\text{”}:2, \text{“}S\text{”}:0, \text{“}D\text{”}:0$$

The “2” in {“A”: 2} is represented by

$$C(I_{n-N+1}^{n-1} I_n) \tag{4.3}$$

and denotes the number of times “A” has occurred following the prefix “W”, “W”. The probability of the next interaction, given some history, is therefore defined by:

$$P(I_i | I_{n-N+1}^{n-1}) = \frac{C(I_{n-N+1}^{n-1} I_n)}{I_{n-N+1}^{n-1}} \tag{4.4}$$

4.7. Implementation and performance

It is conceivable that, depending on the complexity of the prediction algorithm, an excessive amount of time to compute predictions may be expected. This delay, no matter how small or great, will introduce and contribute to the overall IL experienced by the user. It is therefore critical that prediction modules be carefully evaluated for performance before they are integrated into an IRR system.

This section describes two implementations which were evaluated for N-Gram storage and performance: buffer-based and a custom dictionary-based. For each approach, performance was evaluated by generating 10,000 random interactions and timing how long each prediction took to be produced.

4.7.1. Buffer-based

In the buffer-based implementation of the N-Gram model, each new observed interaction was appended to the corpus, which were stored in a list data structure. With this approach, the pseudo code for the prediction algorithm is as follows:

For each interaction:

1. Get the last N interactions (prefix)
2. $C(X)$ = Count number of times prefix occurs within corpus
3. $C(Y)$ = Count number of times prefix + latest interaction occurs within corpus
4. $P(I_i | I_{n-N+1}^{n-1}) = \frac{C(X)}{C(Y)}$

5. Add interaction to corpus

The model was evaluated in two different modes: trained and untrained. In the trained mode, the training data is added to the corpus. However, in the untrained mode, this process is skipped, and the corpus is empty at the time of the first prediction. When an interaction is performed, a prefix is generated. The corpus is then searched for the prefix and each occurrence is counted. In addition to this, the immediate-next interaction after the matched prefix is noted and counted.

After training the model on 4000 of the total 5000 interactions, the remaining 1000 interactions were used as testing data. They were fed into the application one at a time and before each interaction, a prediction was made. The model was then updated with the interaction and the result of the prediction (correct/incorrect) was recorded.

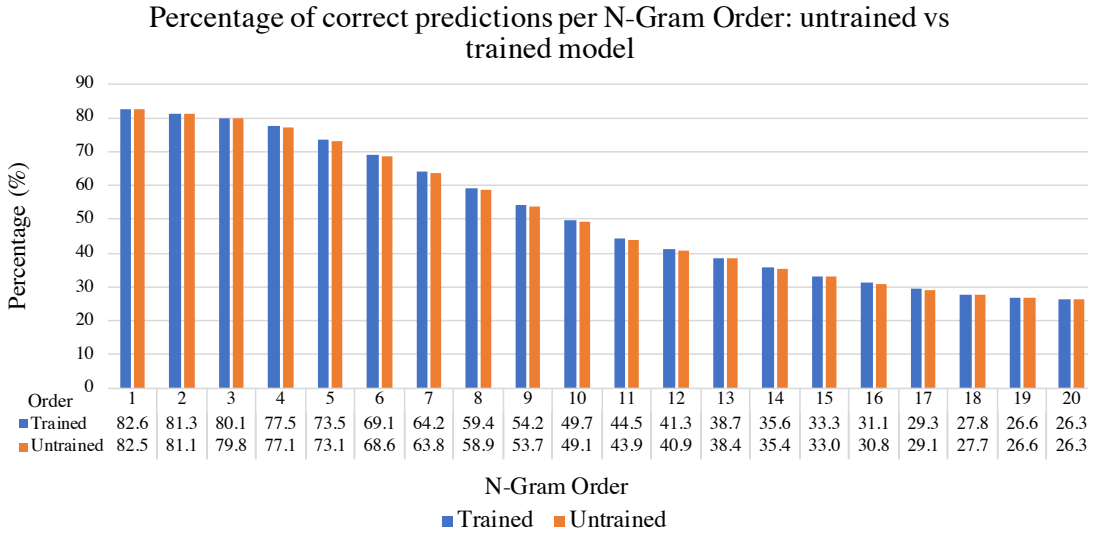


Figure 4.2 Trained vs untrained N-Gram model.

The trained model was evaluated 20 times and with each run, the order was increased. The results were logged at the end of each experiment. The same set of experiments were then repeated using the untrained model. Figure 4.2 shows that as the N-Gram Order, or the amount of history used, increases, the number of correct predictions decreases, regardless of whether or not the model is trained. It can also be seen from the figure that there is very little difference between the trained and untrained model in terms of the percentage of correct predictions: the untrained model performs slightly (a fraction of a percent) better, which is likely due to bias in the training data.

During development, it was noticed that after extended periods of use, the length of time required to compute predictions increased. After investigating, the problem was found to be with the design of the model storage: with each new interaction, the corpus list grew longer and continued to do so for as long as the user used the system. In turn, this caused the time required to perform pattern-matching to increase (as the model had to regenerate the N-Grams for each

prediction), which was therefore ultimately impacting latency. Of course, the length of interactions collected could be limited such that new observations cause old observations to be removed. However, removing previous interactions from the corpus would result in less observational data and would likely negatively impact the ability of the model to match new patterns. Figure 4.3 illustrates the increasing computation time using the approach just described. Note that the two pronounced "dips" are the result of using shared instances on AWS, where for a short period of time, Amazon instances were prone to sudden spikes in CPU utilisation. Nevertheless, the dips are minimal, measuring less than 2 milliseconds in magnitude.

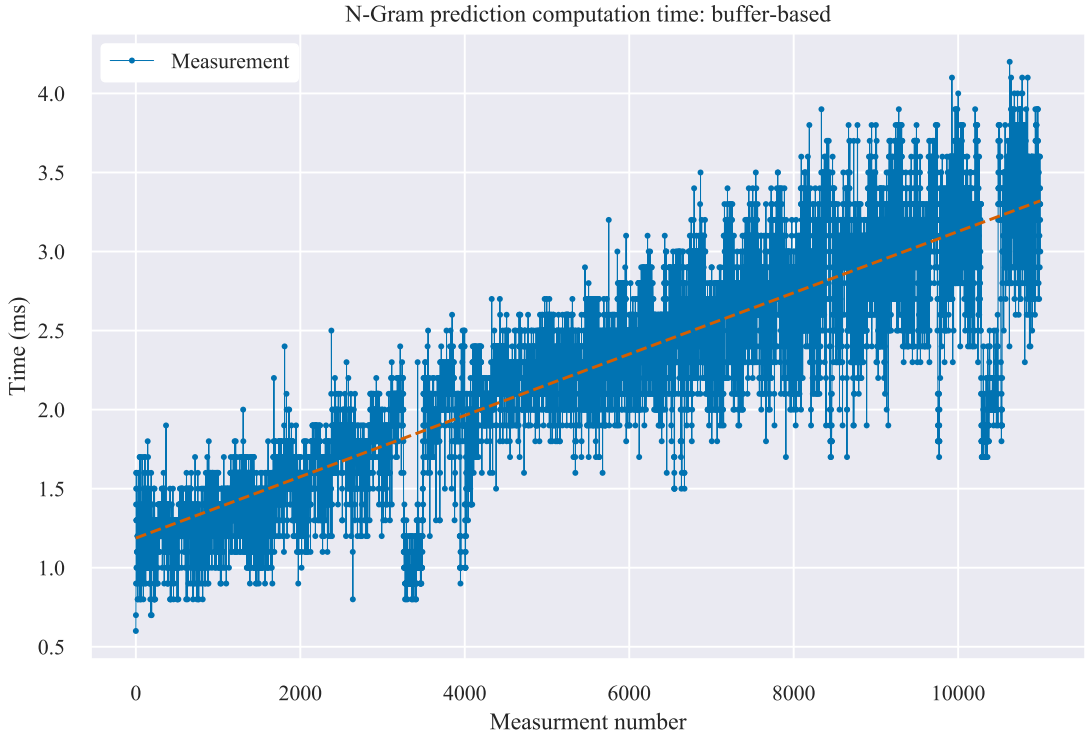


Figure 4.3 Computation timings for N-Gram prediction using a buffer.

4.7.2. Dictionary-based

In order to improve the performance of the prediction model, a new data structure was chosen. Dictionaries are highly efficient data structures and according to Microsoft, "*Retrieving a value by using its key is very fast, close to O(1), because the Dictionary<TKey, TValue> class is implemented as a hash table*" [85].

The data structure was redesigned such that the model itself is a dictionary, whose "key" is the prefix, and "value" is a class object called "Gram". The Gram class contains a dictionary called "Observations", whose "key" is an "observation" and "value" which is a count of the number of times that observation has been encountered. An "observation" is an interaction recognized immediately following a prefix. In other words, if the corpus was "w", "w", "a" and the prefix was "w", "w", the observation would be "a". The Gram class has a function named "Results", which calculates the total number of Observations for its parent prefix and then returns a new

dictionary of with the same keys as Observations, but with probabilities as values and with the results ordered descending-wise.

When an observation (interaction) is added to Observations, it is used as a key and its value is set to 1. One is used since probabilities for future interactions must be calculated, and to do that, prefix counts must be divisible by observation counts, which cannot be done if the value were to be set to 0. If the observation already exists in Observations, its value is incremented by 1.

The pseudo code for the model training algorithm is as follows:

Algorithm 2 Training and predicting user N-Gram model

```
1:  procedure TRAIN
2:      Get unique interactions
3:      for i = 0; i < length(training data - Order); i++ do:
4:          Create a Key
5:          if (key not in model<key,prediction>) then
6:              Create empty Gram object and initialise Observations with unique interactions
    (keys) and the (value) 1
7:              Add a new {Predict : gram} pair to the dictionary model
8:          Get interaction immediately after key (e.g. trainingData[count + Order])
9:          Find prediction in model and increment corresponding interaction value
10: procedure PREDICT
11:     Get Prefix
12:     if model does not contain Prefix then
13:         Create empty Gram object and initialize Observations with unique interactions (keys)
    and the (value) 1
14:         Add a new {Predict : gram} pair to the dictionary model
15:     Retrieve prediction object from model with matching Prefix
16:     for all Observation do
17:         Calculate probability with: observationvalue / sum(allobservationvalues) * 100
18:     Order observations according to their probability (in descending order)
19:     Return observation with highest probability
```

Figure 4.4 explains the model structure.

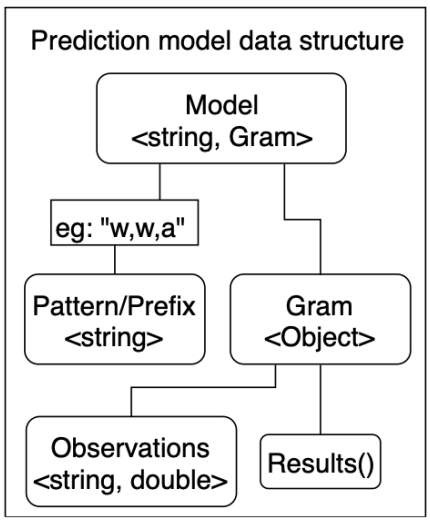


Figure 4.4 Dictionary-based data structure used for N-Gram prediction model.

The same random data used in the previous experiment and shown in Figure 4.3 was used to evaluate the new data structure implementation, the results of which are shown in Figure 4.5. As can be seen, the new dictionary-based data structure significantly improves performance and prediction computation times no longer increase with use. On average just 0.1ms is required to generate a prediction.

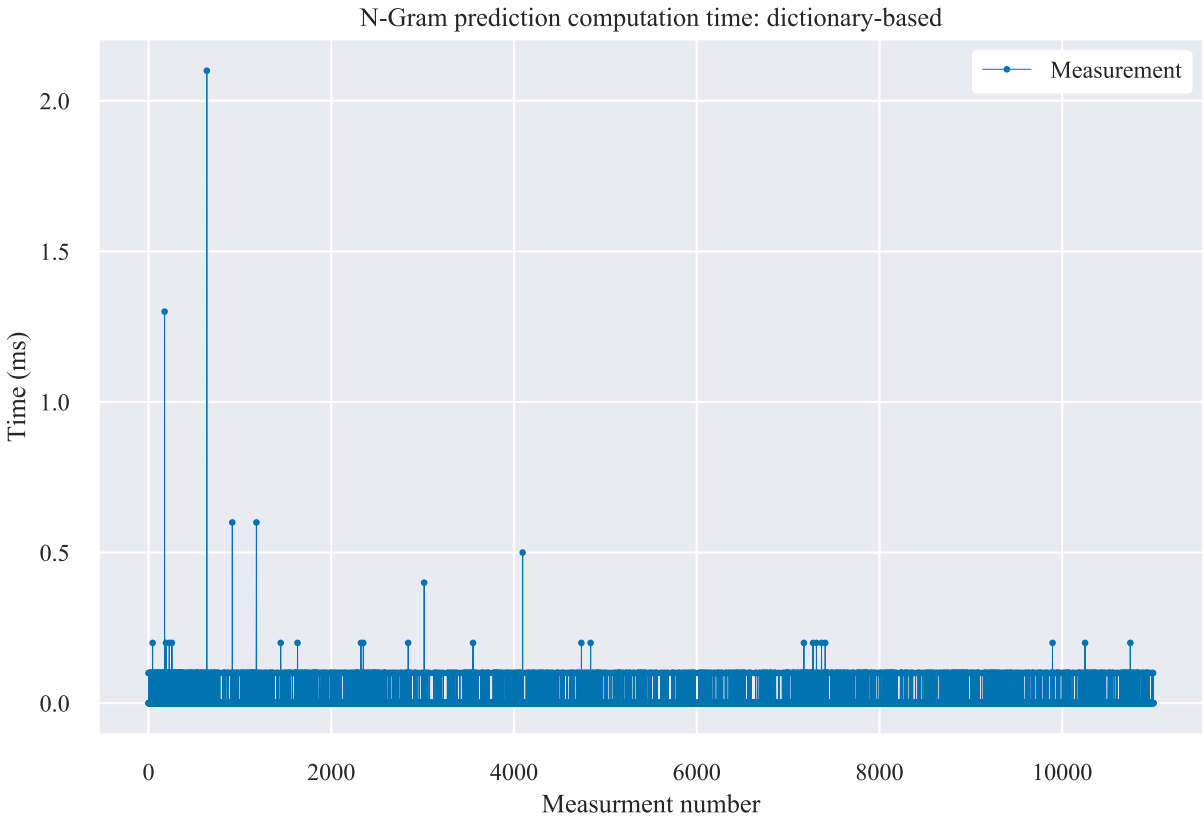


Figure 4.5 Computation timings for N-Gram predictions using a dictionary-based approach.

4.8. Summary

As described earlier, N-Grams have been used by AI opponents in video games for predicting user interactions and providing counter moves with a lot of success. However, it remains to be seen whether those previous successes can be realised in the context of IRR systems, and whether or not N-Grams can be used to aid in the reduction of IL. This chapter has introduced N-Grams, described a model for predicting user interactions as well as demonstrated and evaluated two implementations.

Prediction accuracy was evaluated at different orders on both trained and untrained models using data collected from 5 randomly generated VE mazes. The results of these experiments indicate that there is very little difference between trained and untrained models, at least in this context. Two implementations were described and investigated for prediction: buffer-based and custom dictionary-based. The two implementations were evaluated for performance by randomly generating a set of 10,000 interactions and recording how much time was required for each prediction to be produced. The results indicate that the dictionary-based approach provides significant performance benefits, requiring approximately 0.1ms per prediction.

During testing, it was found that the more frequent consecutive interactions changed, the greater number of incorrect predictions to occur. For example, if a sequence of interactions is "w,a,w,d,s,a,d,a,w", both models struggle to accurately predict the next action, whereas with a sequence such as "w,w,w,w,a,a,a,a,a", the models produce fewer incorrect predictions. This is because N-Grams match patterns within the data and the more erratic the data, the more difficult it is to predict. Indeed, with just 4 possible interaction types, a random set of interactions would reveal an accuracy of just 25%. As the order is increased and more information is used to identify patterns, the model performs worse and worse because its ability to capture patterns is diminished.

Chapter 5. Simulating a Predictive Interactive Remote Rendering System

5.1. Overview

Designing and building an IRR system is a challenging and time-consuming task. Due to their use of Wide Area Networks (WAN), a critical issue, IL, is present in all IRR systems. However, user interaction prediction might offer a way to lower overall IL. This may be achieved by performing rendering ahead of time, and delivering interaction results to the client before, or as close to when, the corresponding interaction has been performed on the client device.
In the previous chapters, we have described how to model and measure (§3) IL, as well as how to predict keyboard-based interactions (§4).

In this chapter, a framework for a PIRR simulator system is described with the purpose of providing a testbed for reproducible experiments, hypothesis testing and evaluation. In addition, this chapter seeks to understand how to integrate prediction into an IRR system and the implications of doing so.

Towards this, the prediction model presented in §4 was built into a separate application (referred to as the prediction module) so that updates and modifications to the model and prediction algorithm could be more managed easily, as well as to enable the prediction module to be experimented with at different locations on the network. The addition of a prediction module into the IRR system introduced various complications, however. For example, raised questions include: how is IL measured when prediction frames are not a direct result of interaction? How are communications between all the components managed? How are correct/incorrect predictions identified by the client application so that a failed prediction frame is not displayed to the user? How should results arriving on the client application (from the server application) be stored and managed, since some results may arrive before their corresponding interaction has been performed? The ability to simulate such a system would make experimentation with system design and prediction modelling simpler, as it would enable developers and researchers to quickly test new or updated system components in a controlled environment and without a large investment of time. Therefore, this chapter focus on introducing a PIRR system simulator and describing its architecture. In subsequent chapters, this simulator is used to evaluate the effects of N-Gram prediction on IL. Later, in §7, a real PIRR system is introduced, evaluated and then compared with this simulator.

5.2. Definitions

1. *Interaction template*: a text file containing interactions – one per line.
2. *Interaction queue<string>*: a FIFO queue used to store interactions read from the interaction template.
3. *Interaction number (integer)*: the number for the current interaction.
4. *ID (string)*: The ID is a unique string representation of the current interaction number, action performed and location. The scheme used is:
   $ID =< interaction_number >_< action >_< pos.x >_< pos.y >_< pos.z >$
5. *Interaction (string)*: This represents the action performed, simulated or otherwise. Only the character representations of the keys pressed as a string, are stored. An interaction can be either W (forward), A (left), S (backwards) or D (right). At the end of each interaction, an interaction number, initially set to 0, is incremented. This approach was chosen so that each interaction can be uniquely represented with the scheme defined above in ID. In the event of an interaction resulting in a previously "visited" location/position, the interaction number will differentiate the past interactions from the new, allowing for the differentiation of otherwise identical interactions.
6. *Position (Vector3: x,y,z)*: The Vector3 is a custom object modelled on the Unity3D class of the same name. It is essentially an object with three float properties, x, y and z which represents the position of the "player". Each interaction effects this property. For example, the position is initially set to 0,0,0 and if a forward interaction is performed, the position will change to 0,0,1.
7. M_c: A message constructed on the client and sent to the prediction module. This message.
8. $M_{pending}$: A "pending" message representing an interaction which has yet to have a result arrive back on the client. This message is identical to M_c except that it contains a Stopwatch.
9. M_p: A message with a predicted interaction and position. This message is sent from the prediction module to the server application.
10. M_s: A message from the server to the client application. This message is identical to M_p however, the Frame property has been updated with frame data and the frame number has been incremented.

5.3. Architecture

The PIRR system consists of three programs, all of which are console applications written in C#: client application, prediction module and server application. Two configurable modes are

available: local and remote. In local mode, the system runs entirely on a single machine. In remote mode, the system executes with the client and server applications on different machines, separated by a network. In all experiments using a live network, the client application is based in Cambridge, UK, while the server application is hosted remotely on an Amazon EC2 instance located in Northern California, USA. Communication between all clients is maintained using RabbitMQ, a popular open-source message broker software. RabbitMQ was chosen due to its easy to use API, the availability of C# bindings (Unity3D uses C#, so integration of RabbitMQ is far easier), the large support community, its maturity and its low-latency message overhead.

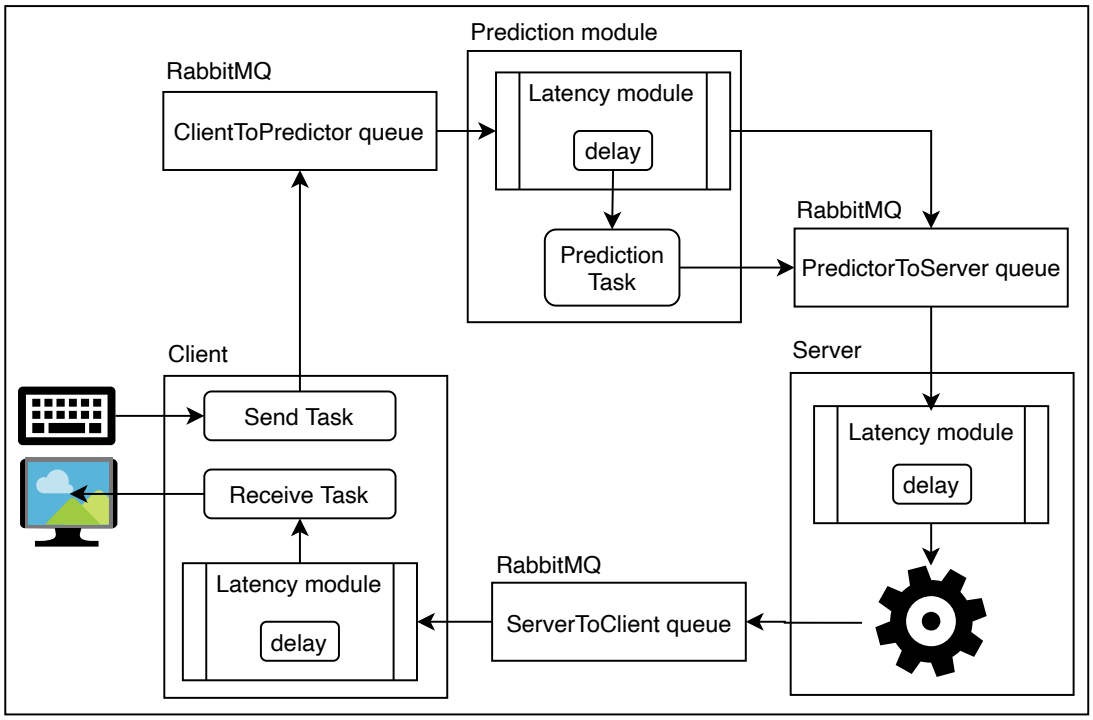


Figure 5.1 Predictive Interactive Remote Rendering (PIRR) simulator architecture.

5.3.1. Client application

The client application is responsible for simulating user interactions, sending them to the prediction module, and for receiving and processing messages from the server application. The client is also responsible for matching received result messages (M_s) with $M_{pending}$ messages stored in the Client Interaction Buffer (CIB) as well as measuring IL.

When the client application starts up, a queue of interactions (referred to as the interaction queue) to be simulated is populated by reading into memory a predefined template, which consists of a single interaction per line. Interactions are performed automatically, and the use of the template ensure repeatable and reproducible experiments. To match the N-Gram model, interactions are constrained to four directions: forward, backward, left and right. However, the system is capable of managing a wider range of (keyboard only) interaction events. These four interaction types are represented by “w”, “s”, “a” and “d” respectively.

Table 5.1 An example of the configuration file used in initialising the system.

```
[General]
client_ip = xxx.xxx.xxx.xxx
server_ip = xxx.xxx.xxx.xxx
predictor_ip = xxx.xxx.xxx.xxx

[Experiment]
run_locally = true
# Interactions can be performed manually or automatically, for experiments.
auto_interactions = true
experimentName = a_description_of_the_experiment
# Latency in one direction
latency = 50
# In some experiments, prediction is disabled.
usePrediction = true
# The Send Delay (SD) of the system
delay_between_interactions = 100
```

After populating the interaction queue, a configuration file is loaded into memory; this configuration file allows for various properties to be set such as whether or not to enable prediction, how much latency to simulate, the delay between interactions and what order N-Gram is to be used. Table 5.1 is an example of a configuration file used to communicate experiment parameters to the PIRR simulator.

Three RabbitMQ queues are then created: clientToPredictor, predictorToServer and serverToClient. The client then establishes connections with the clientToPredictor and serverToClient queues. To ensure all applications within the PIRR system are ready and configured, a temporary task is created and waits for an initialization message to be passed through the system. Once the client application has received the original initialization message, the system is in the ready state and a simulation can begin. The temporary task shuts down and new three tasks are created and started: Send Task, Receive Task and Process Task.

The Send task loops continuously, until shut down. With each iteration of the loop, the following takes place:

> An interaction is dequeued from the interaction queue originally created, initialized and populated at application start-up. A position property, representing a "player" or "camera", is then updated according to the dequeued interaction. In other words, if the dequeued interaction is an "A", the position property is updated to reflect a left movement: $0,0,0 \mapsto -1,0,0$. An ID is then created using the scheme defined in §5.2 and assigned to a new $M_{pending}$ message. A stopwatch on $M_{pending}$ is then started. The $M_{pending}$ is then added to the CIB, where the created ID is used as the key and $M_{pending}$ is the value. A message object used for communicating interactions to the prediction module (M_c) is

created using $M_{pending}$. $M_{pending}$ and M_c are identical, except that M_c does not contain a stopwatch. M_c is then serialized and sent to the prediction module via the clientToPredictor queue. After transmitting M_c to the prediction module, this process then sleeps for 100ms. 100ms was chosen arbitrarily, however this can be set in the configuration file; this delay represents the SD of the system.

The Receive task, like the send task, loops until shut down. The purpose of this task is to receive result messages (M_s) from the server application and make them available to the process task. When a result message M_s arrives on the client application, it is immediately passed to the latency module, which delays the message a specific amount of time before allowing it to be processed. As soon as the message has been delayed the required amount of time, an event is raised and M_s is deserialized and added to a dictionary where the "key" is $M_{s.ID}$ and the "value" is M_s. This dictionary is known as the Client Frame Buffer (CFB) and is used to store results arriving from the server application. The CFB is a crucial component: if predictions arrive on the client before their corresponding interactions have been performed, they must be stored for use at a later point in time. Without the CFB, predicted interaction results may be lost (if not stored) or displayed before any action has been performed (if shown immediately).

The Process task is executed when a new M_s message arrives on the client application. Immediately after arriving, a check is performed to determine if M_s has an ID (no ID for non-interaction generated frames). If M_s does have a valid ID, the corresponding $M_{pending}$ is extracted from the CIB using this ID. The dummy frame from M_s is then presented and the stopwatch associated with the $M_{pending}$ message is stopped. IL can then be measured. It is possible, however, that no corresponding $M_{pending}$ exists in the CIB. The reason for this is that if prediction is enabled, the a predicted M_s may have arrived before the corresponding interaction has been performed. The frame from M_s is not displayed in the simulator system.

5.3.2. Interaction-Result matching

Initially, it was expected that a simple integer would be sufficient for identifying which M_s is a direct result of M_c. However, it was soon determined that the approach is not suitable for measuring IL when a prediction module is used. This is because the prediction module will generate n possible future results and deliver them to the client, hopefully before the corresponding interactions have been performed. As a result, the prediction module would produce predictions but would be unable to assign a valid ID to them, since that ID must come from the client. Additionally, when an interaction arrives on the prediction module, it would be impossible to determine whether or not a previous and correct prediction has already been transmitted to the server application. This is important because without this knowledge, the prediction module would send duplicate messages to the server application, which would result in wasted bandwidth as well as critical server render time and system resources.

In order to solve the problem of interaction-result matching introduced by the addition of a prediction module, the prediction module must be able to generate an ID (described in §5.2) that will exist in the CIB in the future. Following an example of VE systems, the user can navigate and therefore has a "position" attribute. This position attribute, as well as the interaction performed and the interaction number, can therefore be used to define a unique and predictable ID which can be generated by the prediction module in advance. Additionally, the interaction number serves as a useful property for discerning past and future results: if the interaction number is less than the current one, the message is old and can be discarded, whereas if is identical or greater than the current interaction number, the message must be consumed either now or possibly in the future (if the prediction is correct). Once a M_s has been matched with a $M_{pending}$, both are removed from their respective dictionaries: CIB, CFB. If no corresponding $M_{pending}$ exists in the CIB, M_s may be required in the future and is therefore left in the CFB. A cleanup process ensures that all expired messages in the CFB are discarded.

5.3.3. Prediction module

The responsibility of the prediction module is to accept incoming M_p messages from the client application, use the associated interaction to predict future interactions, and forward those predictions (M_p) to the server application where they can be processed, rendered and sent to the client application. The prediction module can be either integrated into the client application, integrated into the server application, or stand as a separate application, as depicted in Figure 5.2

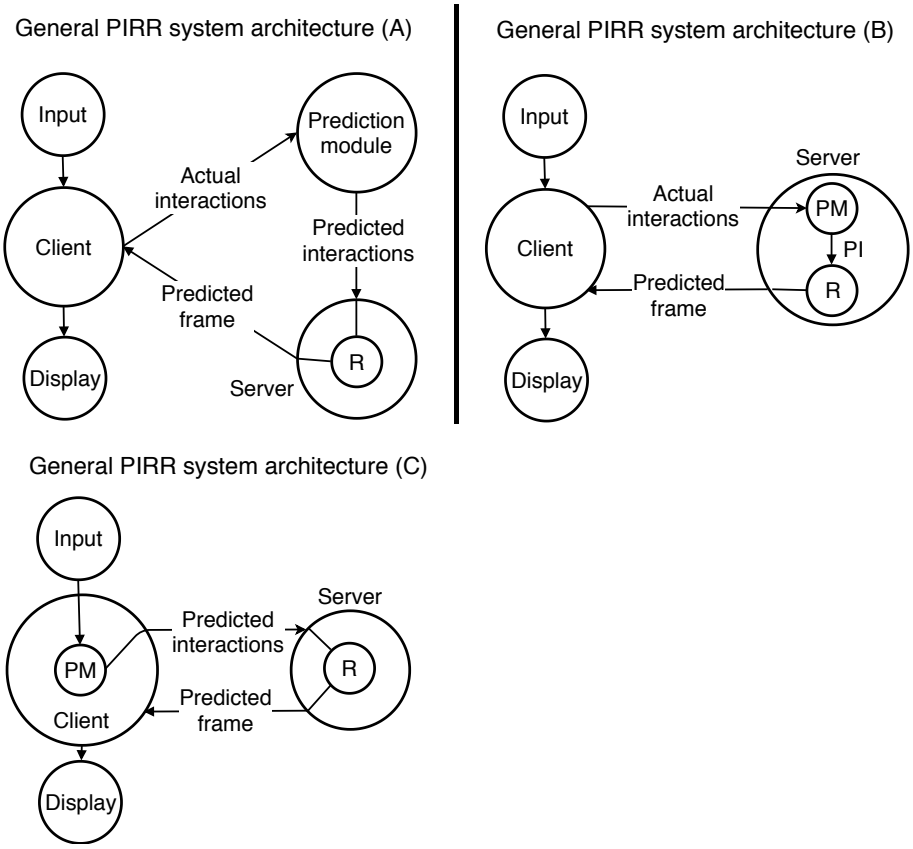


Figure 5.2 Various prediction module integration configurations.

By integrating the prediction module into either the server or client application, a number of issues might arise. For example, the prediction module will itself consume resources and may need a dedicated machine; if the prediction module crashes or slows down, the PIRR system may become slow and in the worst case, unresponsive. Another, more restrictive issue is that by integrating the prediction module into an application, it becomes difficult to modify and maintain. The prediction module was therefore designed as a separate application for two main reasons:

Scalability. The prediction module will itself consume resources. The more sophisticated the prediction algorithm, the more resources it will consume. Having a separate prediction module allows for it to be hosted on a server different to the one hosting the renderer or the client, thereby preventing it from impacting other PIRR system components.

Portability. If the prediction module is built into the server or client applications, there is no way to change or move the prediction algorithm without modifying the application it is built into. Having a separate prediction module means that the prediction algorithm can be measured without impacting the host application. If the rendering software is closed-source, the prediction module must be separate.

On startup, connections to two RabbitMQ queues are established: clientToPredictor and predictorToServer. Messages from the client arrive in the clientToPredictor queue. The prediction module listens for a system initialization message and when received, processes it and forwards it to the server application. Two tasks are then created and started: *Receive Task* and *Process Task*.

The receive task simply loops and listens for the arrival of M_c messages. When a message arrives, it is added to a FIFO queue for processing. This task loops until shut down.

The Process task loops until shut down. On each iteration, a FIFO queue (into which M_c messages are added by the receive task), is inspected. If a message exists, it is dequeued, one or more predictions are created, and those predictions are sent to the server application via the RabbitMQ predictorToServer queue. Prediction integration is described in further detail in §5.5.

5.3.4. Server application

The server application is very lightweight in that its only purpose is to receive M_p messages from the prediction module, simulate rendering and transmit the resulting M_s message to the client application.

On startup, connections to two RabbitMQ queues are established: predictorToServer and serverToClient. As with the prediction module, the server application awaits an initialization message, which, once received, is used to set some local properties. Specifically, the amount of

NL to simulate, the server application ID (a simple integer) and the amount of rendering time to simulate are configured. Since actual rendering is not performed, a dummy image (size: 640 x 480) is read from disk to memory. When a message M_p arrives from the prediction module it arrives in the RabbitMQ predictorToServer queue. An event is then raised, and the message is passed to a separate task, called Process.

The Process task loops continuously until a shutdown message is received. When a message is received, the server application simulates rendering by sleeping for some amount of time, which is set to 30ms. When 30ms has elapsed, a new message M_s is created. The message is identical to M_p except for the frame property, which is has the dummy frame set. Finally, M_s is serialized and sent to the client application.

5.4. Latency simulation module

In order to allow for controllable and reproducible experiments with the PIRR simulator a latency module was packaged into a Dynamic Linked Library (DLL). The latency module is used by each of the three applications in the PIRR system and in each case, is only placed at the point at which the messages arrive. Figure 5.3 illustrates the positions of the latency simulator for each of these applications.

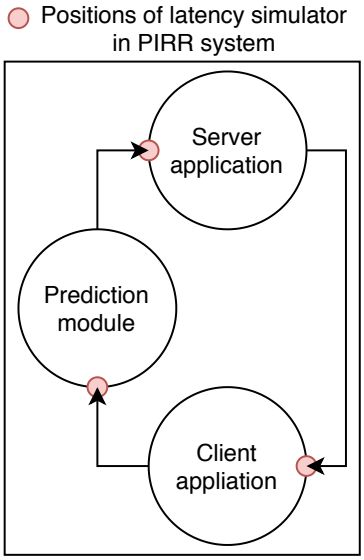


Figure 5.3 Positions (red circles) of the latency module within the PIRR system.

To use the latency module a single method is exposed: “Delay(message, delay)”. This method is used to delay a specific message for a certain duration and an event binding called “MessageReady” is exposed. The application using the latency module must bind to this event: when it is raised, the now-delayed message will be available for further processing.

5.5. Implementation of single-track prediction

Prediction of user interactions takes place in the Process task on the prediction module. Within this loop, the first step is to determine whether or not an interaction message has arrived. When a message M_c from the client application is received by the prediction module, a test is performed to determine if a prediction for M_c has previously been made and sent to the server application. This test involves maintaining a dictionary (simply called "History") of all previously generated and sent messages. If the History dictionary contains a key with an identical ID to that of M_c, a correct previously predicted message has already been generated and sent to the server application. On the other hand, if the test fails, M_c is converted to a M_p without modification (i.e. this is not a prediction – it is simply forwarding the message) and is immediately transmitted to the server application. This ensures that in the event of an incorrect prediction, the client will experience at the worst the full Round-Trip Time (RTT), as if there were no prediction module. The N-Gram model is then updated to incorporate the latest interaction information from M_c for use in future predictions.

While the prediction module waits for a message from the client, the available idle time is used to continue to predict ahead, sending the predictions to the server application and eventually, to the client. Crucially, this enables the system to take advantage of NL between the client and the server, as well as idle time between user interactions. If no message from the client is available, the module determines whether it is able to predict ahead. How far the module can predict ahead is defined as the Maximum Predictions Ahead (MPA), which is set during system initialization. The number of remaining predictions is initially set to equal MPA, decreases with each prediction transmitted and increments with each M_c received, allowing the system to continuously maintain MPA steps ahead of the client application.

This approach can be thought of as a single-track prediction scheme, where previous predictions and interactions drive the model down a single "trajectory", illustrated by Figure 5.4

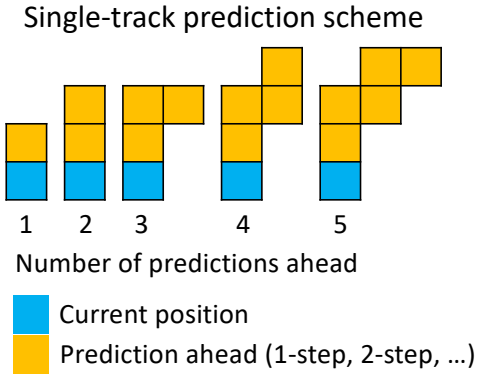


Figure 5.4 Example of single-track prediction.

Since it is likely that, on occasion, the N-Gram model will fail to predict correctly, there is a danger that incorrect predictions may corrupt the model. This is because in order to predict more

than one step ahead, the model must be updated with predictions. If a prediction is incorrect, future predictions will be based off incorrect data. Therefore, in order to remedy this issue, a second "dummy" model is used. The dummy model mimics the main model and is updated with predictions, whereas the real model is only ever updated with real interactions from the client application.

When an incorrect prediction is found to have been made, the dummy model is reset to match the main model, which prevents incorrect predictions from corrupting the main model. Additionally, the number of remaining predictions is reset to equal MPA. If MPA > 0, a prediction is made, and a check is performed to test if an identical prediction has previously been sent. If it has not, a new prediction message M_p is created, serialized and sent to the server application. Once a prediction has been sent, the dummy model is then updated with the prediction, which is then added to the History dictionary.

Finally, it is important to note that by keeping track of all sent messages, the History dictionary serves two purposes: it provides a mechanism for determining whether a correct prediction for M_c has been sent, and to prevent duplicate messages from being sent to the server application which in-turn prevents the wasting of valuable render time (or in this case, simulated rendering time), since Unity3D is only able to produce one rendering at a time.

5.6. Summary

This chapter has introduced a framework for simulating a PIRR system, made of three components: client application, prediction module and server application.

The use of RabbitMQ to manage communications between the applications significantly sped up the development process, as well as opens the possibility for the system to be modified later for multiple rendering simulators, prediction modules or client applications.

The introduction of the prediction module complicated the process of measuring IL. When the prediction module is not in operation, messages can simply be tagged with a GUID or random ID so that they can be linked to render results from the server and easily identified on the client application. However, the introduction of the prediction module results in a situation where, in some cases, responses to interactions are generated before the interaction has been performed. This means that the ability to easily identify the moment a result of an interaction has arrived on the client, is lost. Nevertheless, as described, this can be remedied by applying a carefully considered scheme to the ID of messages being passed through the system. Additionally, a single-track prediction scheme has been described and demonstrates how the prediction module can be used to predict one or more steps into the future. This design is used as the foundation on which future experiments and implementations are built. An overview diagram of this architecture can be seen in Figure 5.1.

Chapter 6. Exploring the Effect of N-Gram Prediction on Interaction Latency Using the Simulator

6.1. Overview

This chapter aims to explore how the use of the implemented N-Gram prediction module impacts IL when using the PIRR simulator described in Chapter 5. The hypothesis is that by predicting user interactions, future results (with frame data) can be generated on a remote server and delivered to the client before, or very nearly after, the corresponding user interaction has occurred. Using the PIRR simulator, the following topics are addressed:

1. How does the introduction of the prediction module affect IL?
2. Does the prefix length *N* (Order), or the "amount of history to include", impact prediction and ultimately, IL?
3. How far ahead should predictions be made and what is the optimal MPA?

Each experiment consists of 634 interactions. These interactions were collected from the IRR system after manually playing the maze as described in §4.5. The number of interactions is arbitrary as they are simply a result of the number of interactions it takes to complete the maze. Six latency levels were chosen for simulation experiments: 0ms, 50ms, 100ms, 200ms, 300ms and 400ms. These latency values were chosen for the following reasons:

Table 6.1 Simulated latencies used in all experiments, and the reasons for being chosen.

Latency (ms)	Reason for choice
0	For testing base latency of system.
50	Suitable IL ranges for IRR systems.
100	
200	Upper limit of IRR system usability with regards to IL.
300	Excessive IL – unacceptable for most IRR systems.
400	

Each simulation runs for approximately 60 seconds and each is run five times. All IL measurements were collected using the approach described in §3.4.3. The results presented in graphs are averages of those five runs, unless otherwise stated. In all experiment runs, the simulated render delay was set to 30ms. A separate program was created to set system configuration parameters, launch the PIRR simulator and to deposit results in a suitable location

Table 6.2 Config file modification for Maximum Predictions Ahead and N-Gram Order control.

```
[Experiment]
# Previously described properties
ngram_order = 1
MPA = 1
```

on disk. Finally, the configuration used at system startup was modified such that the N-Gram Order and MPA could be adjusted for each simulation experiment:
Finally, in some plots, a rate of increase line is shown. This line is also referred to as the best- fit regression line and is given by the equation:

$$line = mx + b \tag{6.1}$$

where m, the slope, is calculated with:

$$m = \frac{\overline{x} * \overline{y} - \overline{xy}}{\overline{x}^2 - \overline{x^2}} \tag{6.2}$$

and b (the y-intercept) is calculated by:

$$b = \overline{y} - m * \overline{x} \tag{6.3}$$

6.2. Experiments

6.2.1. Base IL

It is important to know what the base IL of the simulator is so that the proportion of any latency due to factors such as poor system design choices and/or coding inefficiencies can be identified. To measure the base IL of the system, the simulator was run without simulated NL and without prediction. Results from all simulations were then averaged horizontally (grouped according to interaction number) to form a final dataset which showed that the mean base IL for the simulator is 32.28ms, which is depicted in Figure 6.1.

This experiment was then repeated an additional five times for each of the remaining latencies from Table 6.1 and the average IL was recorded on each occasion, thus providing insight into the base operating IL of the system at various levels of introduced latencies. With each experiment, the NL and render delay were subtracted from the total mean recorded IL, yielding the remaining average amount of system delay. Figure 6.2 shows that other than the introduced delays, the base system latency averages 2ms to 4ms.

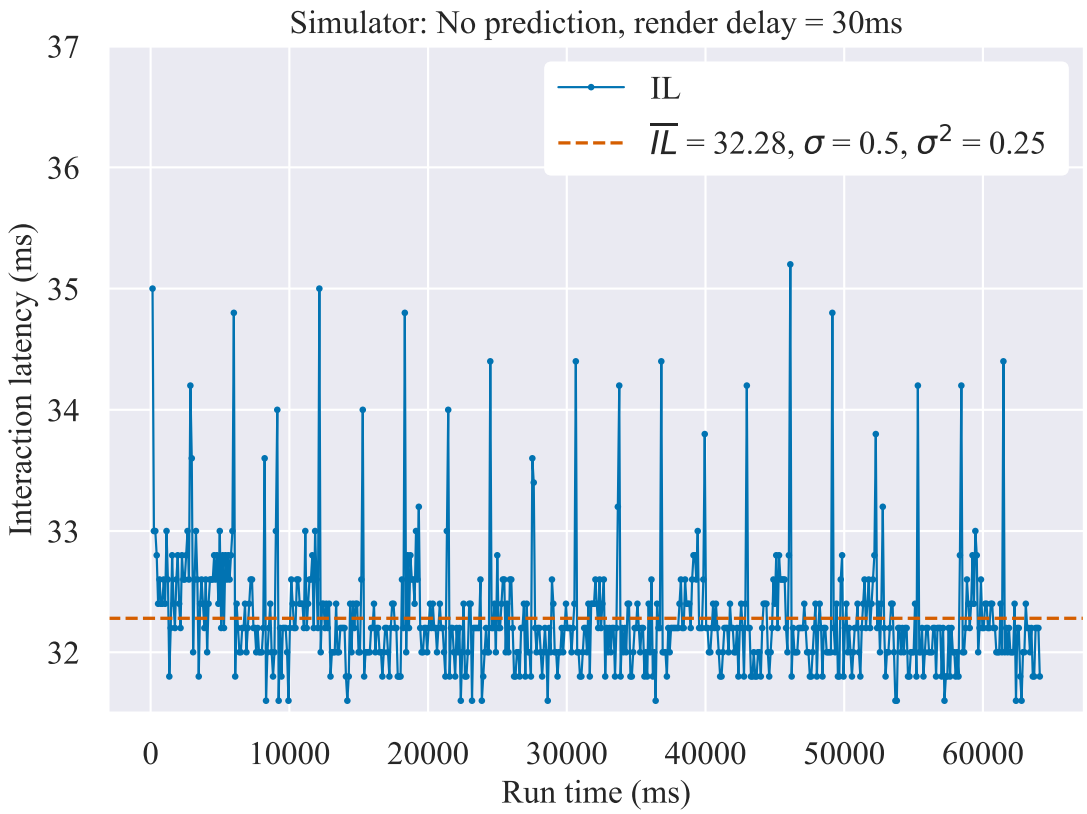


Figure 6.1 Base interaction latency of simulator.

Figure 6.2 confirms that the simulator is operating as expected: the only delays introduced to the system were the 30ms simulated render time and the simulated NL values described in Table 6.1. The measured system latency was found to be a negligible 3ms, calculated as the difference between the measured IL and the sum of the introduced NL and the simulated render delay.

Next, the base IL for the system running over a live network (the internet) was then measured. Using the Windows Ping utility, it was found that the mean NL between Cambridge, UK and an Amazon EC2 server located in Northern California, US was 171ms. After running identical experiments (with no simulated NL, however) the average recorded IL was found to be 216.96ms. Figure 6.3 illustrates raw IL measurements over WAN.

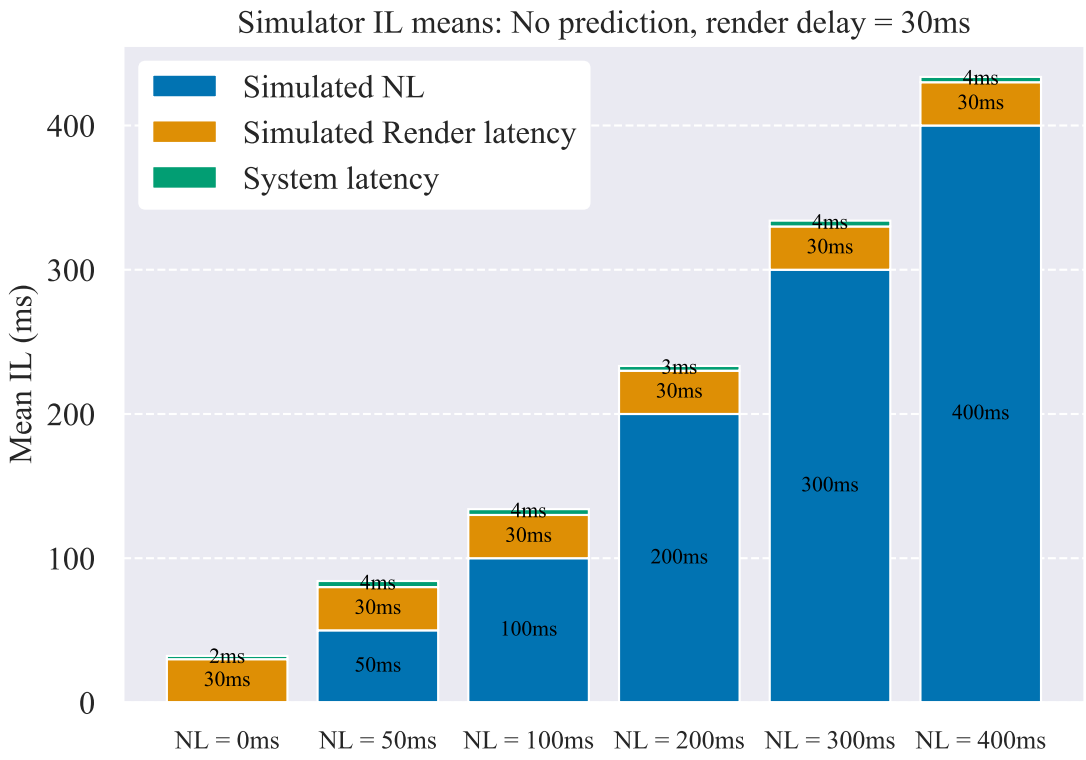


Figure 6.2 Simulator mean base IL at various simulated latencies.

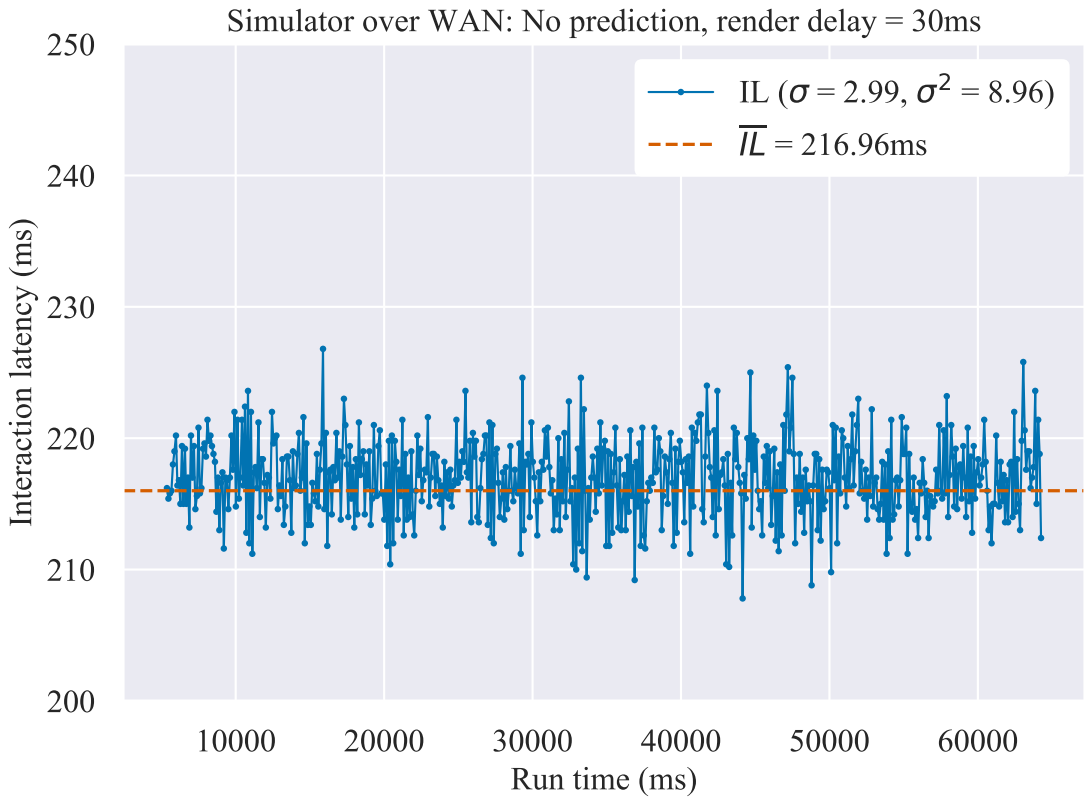


Figure 6.3 Base simulator interaction latency measured over the Internet.

To determine the ability of the simulator to emulate the same network conditions when running over WAN, it was again run five times using LAN but instead of injecting one of the latencies described in Table 6.1, a proportional amount of latency to WAN was simulated. In this case, NL

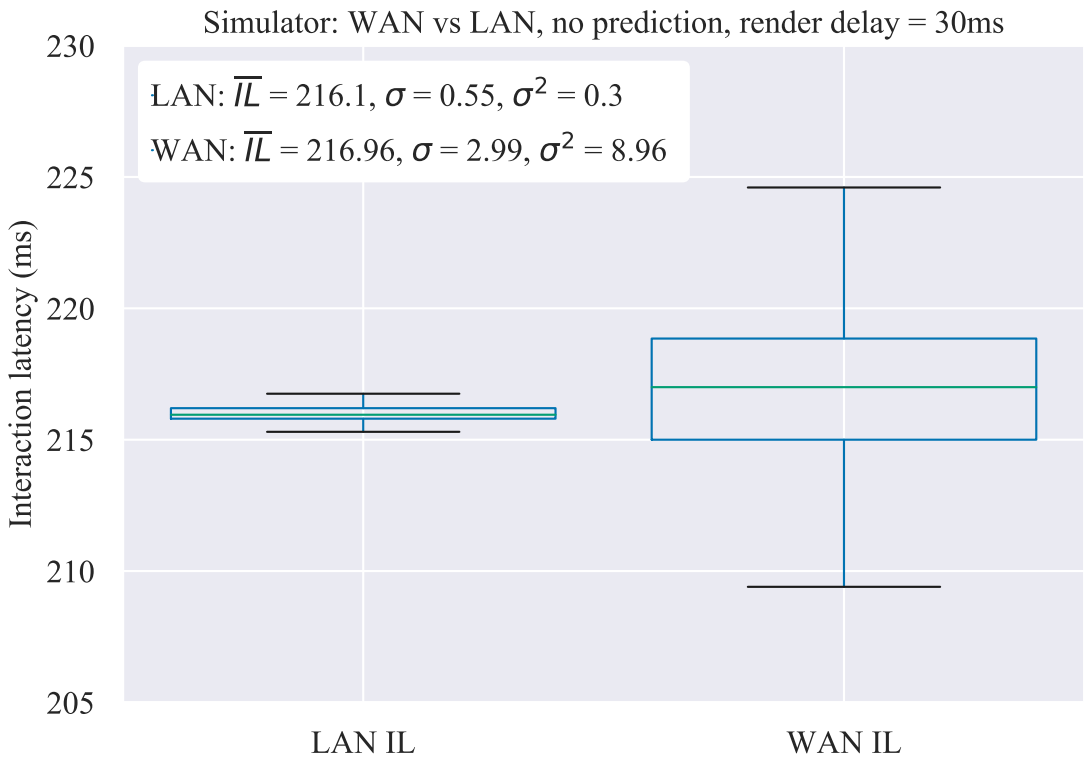


Figure 6.4 Simulator IL boxplots: LAN vs WAN.

was measured (using Ping) to be 171ms between Cambridge and Northern California and therefore 171ms of latency was simulated. No prediction was used.

When IL measurements collected from the LAN experiment runs are compared with those collected from WAN, as in Figure 6.4, it can be seen that there is a difference of just 0.35ms between their means. However, when running over WAN, the data varies a lot more and this is perhaps better illustrated in Figure 6.5. This variation is likely due to the general WAN fluctuation. In terms of the simulator, the slight variance is due to the use of Thread.Sleep() for creating delays, used throughout the system; for example, the latency simulator or to simulate a rendering delay.

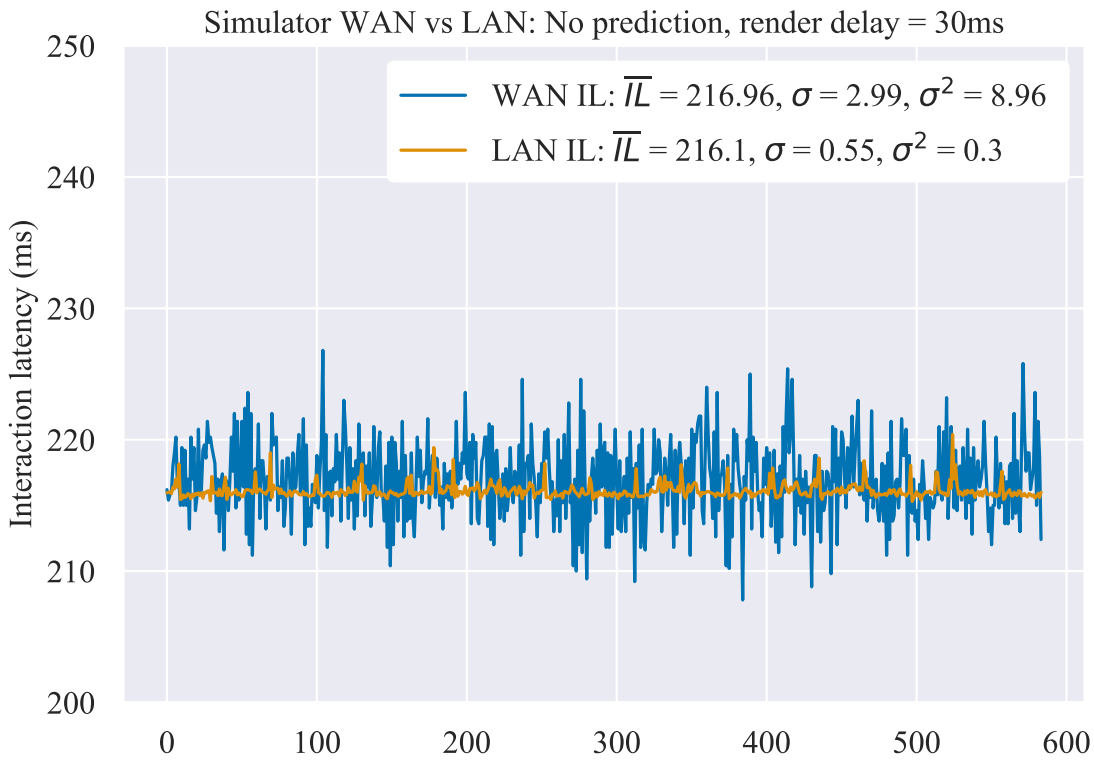
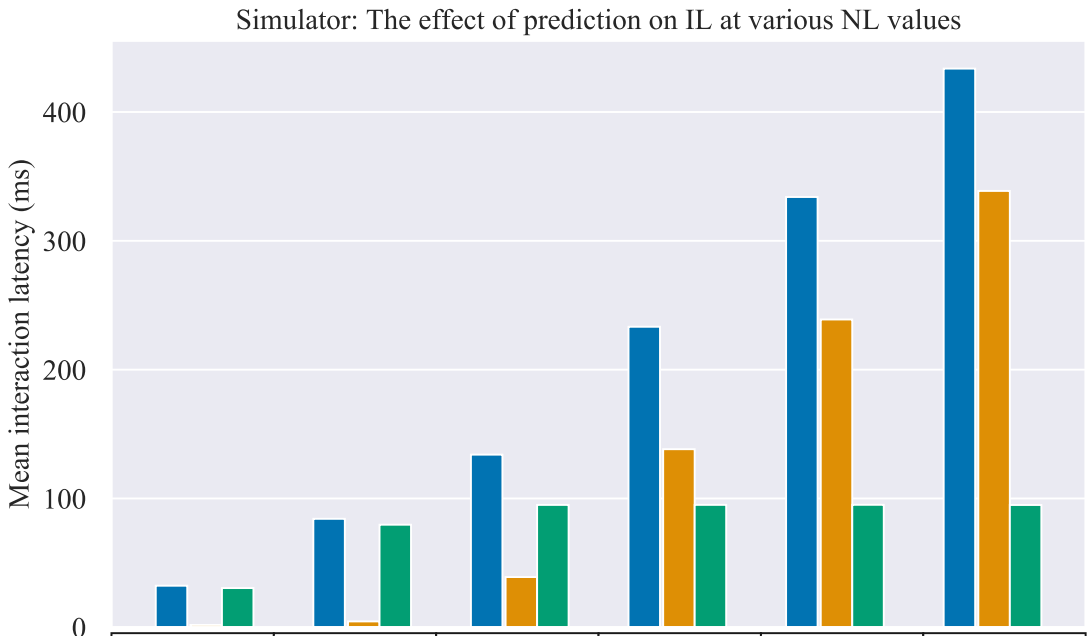


Figure 6.5 WAN IL vs LAN IL using simulator.

6.3. The effect of user interaction prediction on IL using the simulator

By enabling the prediction module, the PIRR simulator is expected to deliver results to the client application before (or very near to when) the corresponding simulated interactions have been performed. Therefore, the aim of the following set of experiments is to understand whether or not there is any notable difference between when operating the simulator with prediction vs without it, and to confirm the expectation that overall IL will be reduced when using prediction. The simulator was run for each of the latencies stipulated in Table 6.1. For all experiments in this section, the N-Gram Order was set to 1, only 1-step ahead prediction was used (MPA = 1), and the model was untrained. Each experiment was performed five times, averaged, and the collected results were then compared with their corresponding non-predicted results and are presented below in Figure 6.6.

Figure 6.6 illustrates that in all experiments and regardless of whether or not prediction is enabled, IL increases proportionally with the amount of simulated NL injected. However, when prediction is enabled, the effects of the prediction module become clear: IL is lower with prediction (P) than without prediction (NP). Interestingly, the difference ($delta$) IL measured between NP and P experiments increases until$NL = 100ms$, at which point the delta remains relatively constant. This may be an indication that a greater number of steps ahead need to be predicted in order to offset the substantial NL.



NL	0ms	50ms	100ms	200ms	300ms	400ms
NP	32.28	84.21	134.03	233.3	334.04	433.7
P	1.82	4.57	39.02	138.26	238.95	338.75
delta	30.46	79.64	95.01	95.04	95.09	94.95

Figure 6.6 The effect of prediction on Interaction Latency using the simulator.

Inspecting the data more closely, it can be seen that IL is nearly entirely eliminated. For repetitive interaction patterns, IL falls to zero and only spikes when an incorrect prediction is encountered. Take for instance Figure 6.7 where a simulated NL of 50ms was introduced using a configuration with MPA = 1 and N-Gram order = 1: while the mean IL is 4.59ms, spikes of IL are observed when an interaction changes from one type to another. For example, if a sequence of forward (W) interactions are observed, followed by a change of direction (e.g. A, for left), the prediction algorithm may fail and the full RTT of the system will be exposed to the user. This is clear when inspecting Figure 6.7: the majority of the IL spikes are around 80ms, which is expected given the injected 50ms NL and 30ms simulated render delay.

The same experiments were then performed over a real network, between Cambridge and California. First, the NL between Cambridge and California was measured using the PING utility, done so at three points in time: morning, afternoon and evening. This was done to counter network congestion and throttling associated with peak-hour internet traffic. NL for these experiments was found to be, on average 188ms.

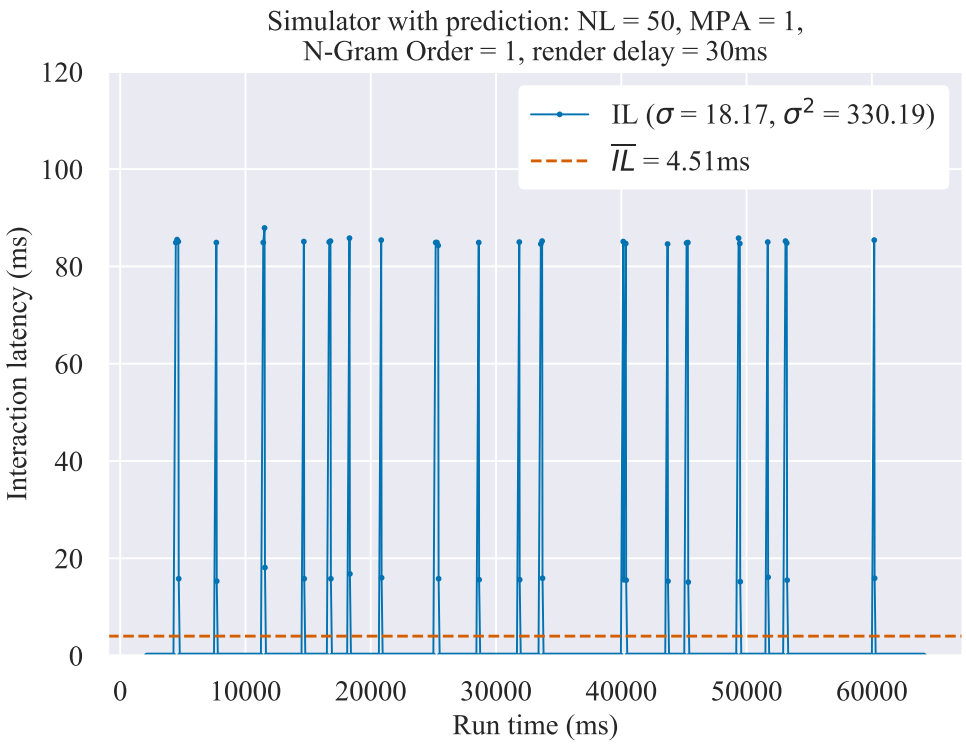


Figure 6.7 Raw data from a single run of the simulator.

The server application was then transferred to, and executed on, the remote Amazon EC2 instance located in Northern California USA, while the client and prediction applications executed locally on a MacBook Pro in Bootcamp mode based in Cambridge, UK. Three experiments (run morning, day and night) were then run using the IRR system: no prediction was used in these experiments.

From Figure 6.8 we see that the mean IL when running the IRR system over WAN is 216.96ms. However, when we enable prediction, setting both N-Gram Order and MPA to 1, and run the system again, IL drops to an average of 134.35ms. This demonstrates that the positive impact of prediction on IL, as can be seen in Figure 6.9. However, as can also be seen in Figure 6.9, large IL spikes occur - again due to mispredictions. Nevertheless, the majority of IL is reduced.

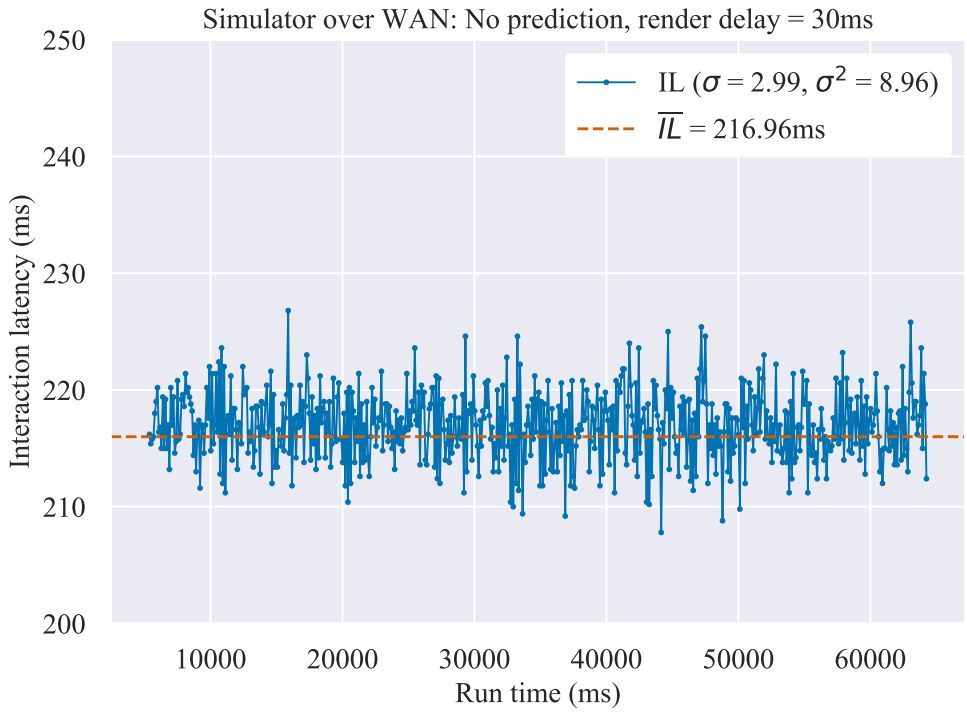


Figure 6.8 Simulator with no prediction, run over WAN.

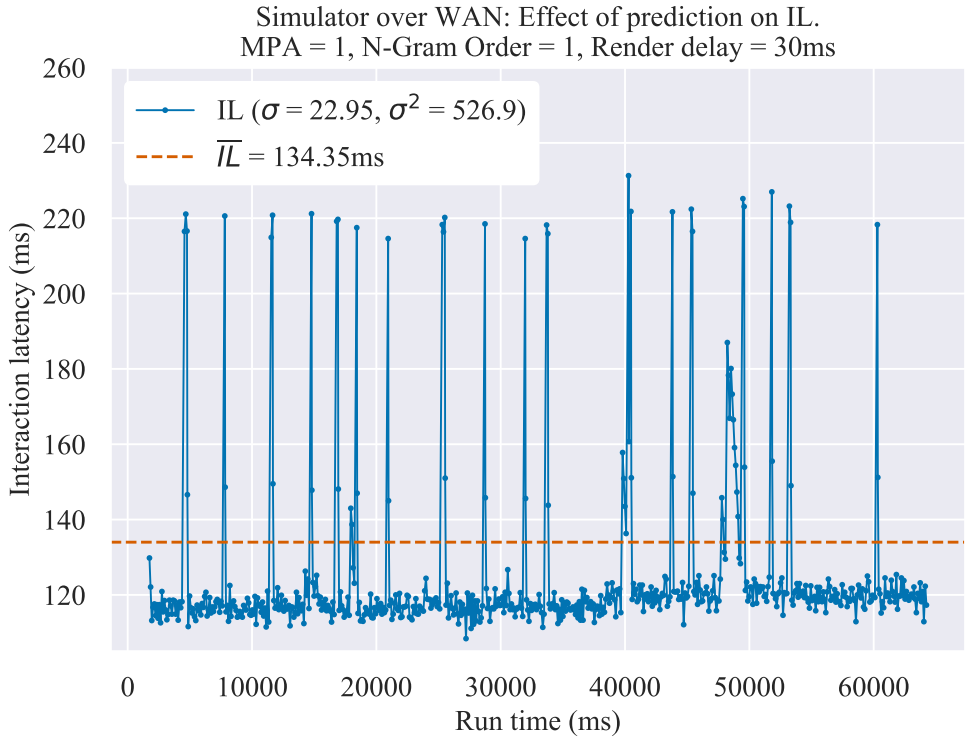


Figure 6.9 Simulator with $MPA = 1$ and N-Gram $Order = 1$, run over WAN.

6.3.1. The effect of N-Gram Order on IL

Earlier results (Figure 4.2) showed that prediction accuracy decreases with higher N-Gram Orders. The hypothesis is therefore that the amount of IL masked by the prediction module will decrease as the Order increases. To test this assumption N-Gram Orders of 1 to 5 were tested for each of the latencies presented in Table 6.1. Experiments were conducted locally with simulated

latencies and each experiment was run 5 times. The results of each quintet were averaged, and the mean IL measured for each experiment group is represented by a single point in Figure 6.10. For all experiments, $MPA = 1$.

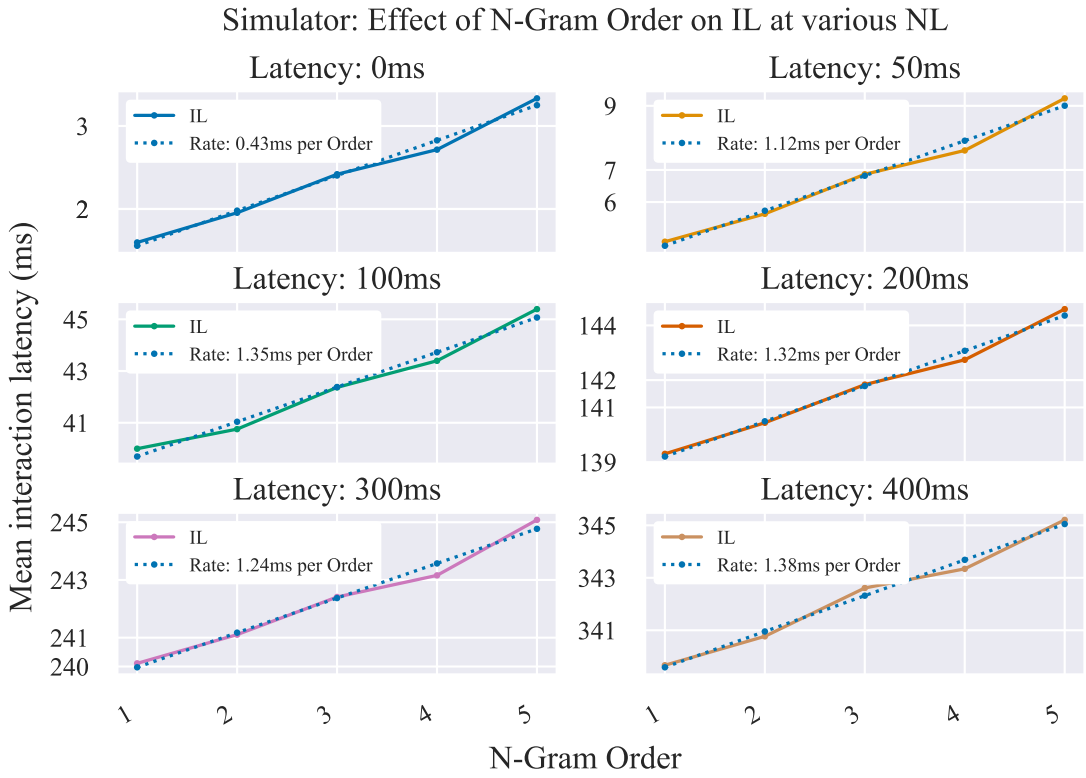


Figure 6.10 Effect of N-Gram Order on IL at various degrees of NL.

The same experiment was then performed 5 times using a WAN, rather than simulated latencies. From Figure 6.10 and Figure 6.11, it is clear that increasing the Order does indeed result in greater IL. The plots also show that on average, IL increases for both simulated and WAN latencies. The larger the N-Gram Order, the more information used as the prefix when predicting interactions. Using too much or too little information will affect prediction accuracy and the algorithms ability to match patterns. The greater the number of incorrect predictions produced, the more often the system will need to make adjustments: when the prediction module incorrectly predicts or receives an interaction for which a prediction has yet to be made, that just-arrived interaction is processed and transmitted to the client and the prediction step is skipped; this results in at least a full RTT latency exposure, but also ensures that minimum disruption to IL is caused, since attempting to re-predict the interaction could result in yet another error, causing further delays.

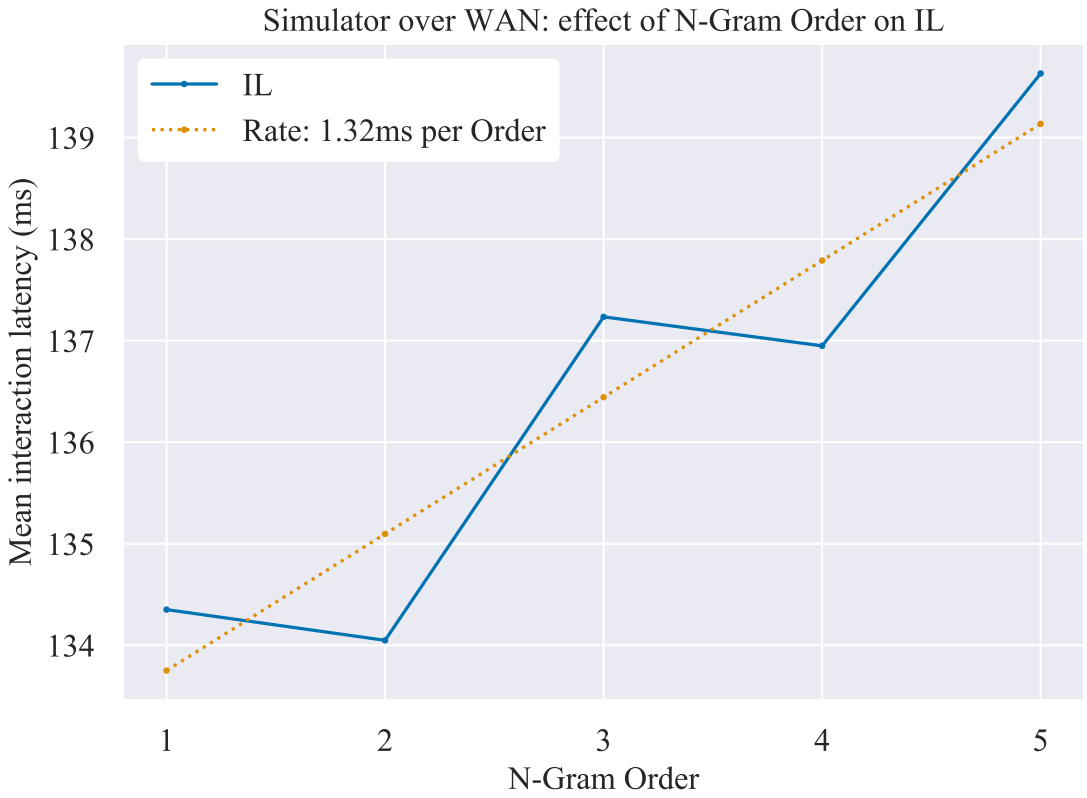


Figure 6.11 The effect of N-Gram Order on IL using the simulator and a WAN.

6.3.2. The effect of MPA on IL

Until now, all experiments involving prediction have used an MPA of 1; that is, only the very next interaction was predicted. In the following set of experiments, the relationship between IL and MPA is explored. It is important to understand this effect because if predicting further ahead results in increased masking of IL, then is it possible to predict too far ahead and if so, how will that affect IL? When a prediction is incorrect, all predictions that follow (until an update is issued) will be incorrect too, since there is no way for the prediction module to know an error has been made until an update arrives from the client application. The Unity3D rendering application can only process one message/prediction at a time and therefore develops a backlog when too many steps ahead are predicted, or when messages from the prediction module arrive too rapidly at the server. Therefore, by predicting too far ahead, the likelihood of wasting bandwidth and computation is increased, since incorrect predictions, if not managed, will cause a backlog to form on the server application as it attempts to process each of the incorrect prediction message.

To evaluate the effect of MPA on IL, MPA from 1 to 10 were tested for each N-Gram Order level (1 to 5), and for each of the 6 latencies described in Table 6.1; These experiments were repeated 5 times, resulting in 1500 simulation runs being performed.

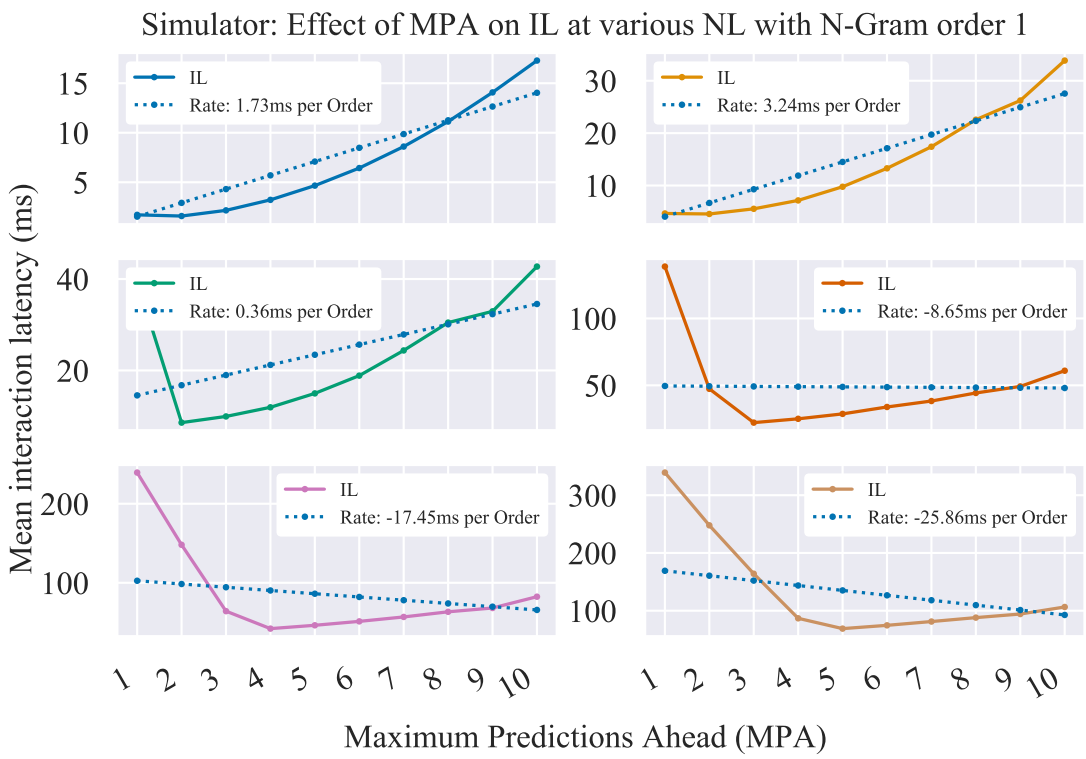


Figure 6.12 Effect of MPA on IL using the simulator.

Above, Figure 6.12 illustrates the results of simulations run with N = 1, for MPA's 1 to 10 at various simulated latencies. The remaining N-Gram Orders (2 to 5) are not presented here. Each plot represents the mean IL recorded over 10 simulations (1 for each MPA) for a given NL. In the first two plots (latencies 0ms and 50ms), IL is at its lowest when $MPA = 1$. This indicates that predicting one step ahead is sufficient and that IL has been reduced as much as possible. Note, however, that in these two plots, mean IL increases with each MPA increase. An identical situation appears in the remaining plots. For NL values 100ms, 200ms, 300ms and 400ms, IL continues to decrease with each MPA increase until a certain point. At this point, MPA begins to increase again.

The reason for this eventual increase is that at a certain point, the number of predictions arriving from the server application cause local buffers on the client to get congested - a backlog forms. Messages within the backlog still need to be processed: the client application must check the message to determine whether or not the predicted result is correct and if so, to apply the accompanying frame to the display. Each increase in MPA results in an additional prediction that will be incorrect (since only 1 prediction can be correct), the amount of time "wasted" also increases per MPA

In addition, as the rendering application processes incoming messages from the prediction module, its own backlog grows because it is only able to process a single message at a time. The result is that at some point, predicting too far ahead causes backlogs to form in both of these places. The same is witnessed with experiments performed over WAN, with a real network - see Figure 6.13.

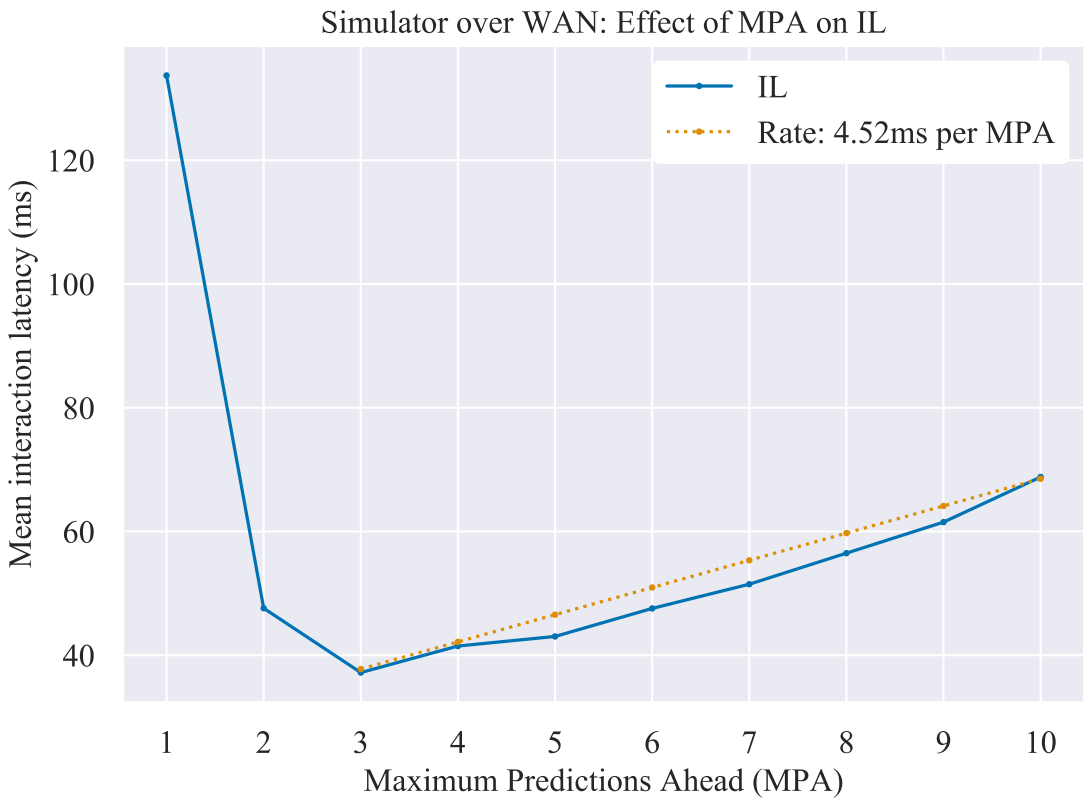


Figure 6.13 Effect of MPA on IL using the simulator over WAN. .

While in a controllable environment, we can test to establish the ideal MPA. However, in a real-world environment, changing network conditions might mean that the ideal MPA will change, too. It is therefore important for the PIRR system to know the ideal MPA before operation and to be able to adjust to changing network conditions.

To understand this, the linear flow of simulator events was modelled and evaluated against sixty experiments: MPA 1 to 10, for each of the NL values described in Table 6.1. N-Gram Order was set to 1 for all experiments. When an interaction arrives at the prediction module, the system will check if the interaction has already been predicted and transmitted to the client. If it has, no further processing of that interaction needs to take place. Instead, the prediction module is free to predict ahead until MPA is reached. If the interaction has not been predicted and sent to the client, it is possible that the previous predictions were incorrect; this causes a backlog on the server application which grows by the render delay (i.e. 30ms) per prediction. However, when correct, a portion of the NL is masked. This is dependent on whether the prediction module is hosted on the client side of the network, or on the server side, which is discussed later in Chapter 9.

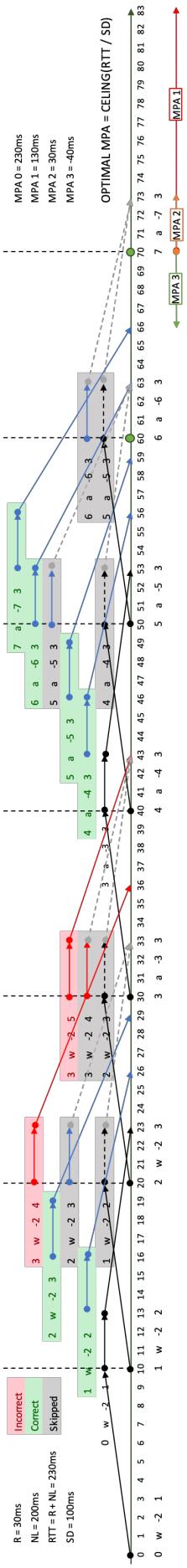


Figure 6.14 Example flow of interaction events through simulator.

Figure 6.14 describes an example flow of events through the simulator. Initially, some constraints are set. The render delay (R) is set to 30, NL to 200ms and SD to 100ms. This results in an RTT (NL + R) of 230ms. The first interaction I_0 performed at t_0 is 0, w, -2, 1. That is, the first interaction number is 0, the action is "w" and the position is -2, 1. The interaction takes 100ms to get from the client to the server (0.5 * NL). When it arrives, the renderer is blocked for 30ms (R). Immediately after producing the render result and sending it back to the client, the next available prediction 1, w, -2, 2 is processed and again blocks the renderer, followed by the prediction 2, w, -2, 3 and then 3, w, -2, 4. At t_1, interaction 1, w, -2, 2 is sent to the prediction module and then to the renderer. When it arrives at t_3:, the system identifies that a correct prediction has already been made and transmitted to the client. This interaction is then skipped. The prediction module then attempts to predict ahead from the interaction 1, w, -2, 2 and finds that 2, w, -2, 3 has already been predicted and sent to the client, and is therefore skipped and the next prediction, 3, w, -2, 4 is processed. At t_4, the interaction 3, a, -3, 3 is performed. Unfortunately, the prediction 3, w, -2, 4 for this interaction was incorrect and the prediction module will not know this until t_5, when the action performed at t_4 arrives. At t_5, the prediction model adapts and begins to make correct predictions, correctly predicting 4, a, -4, 3 and 5, a, -5, 3.

Until now, the system has been predicting just two steps ahead (MPA = 2) with interaction number 1 and 2 arriving on the client 160ms and 90ms late, respectively. Interaction number 3 was mis-predicted and resulted in the full RTT of 230ms IL. At t_6, MPA = 3 is used. As interaction 4 arrives at t_6, it is found that it has been predicted and sent to the client and is therefore skipped. Predictions 1: 5, a, -5, 3, 2: 6, a, -6, 3 and 3: 7, a, -7, 3 are then made. Prediction 1 is found to have already been processed and sent to the client, while prediction 2 and 3 have not and are therefore processed as normal. At t_7, only 30ms of IL is experienced. At t_8, however, interaction 7 experiences zero IL since it arrived 40ms prior to being performed, making MPA = 3 ideal. From this, it can be seen that the optimal MPA is:

$$min(n \in \mathbb{Z} | n \geq \frac{RTT}{SD}) \tag{6.4}$$

Where n is an integer, RTT is the Round-Trip Time of the IRR system and SD is the Send Delay, or the delay between interactions. Using this information, MPA can be adjusted dynamically, taking into account various system delay points such as NL experienced over WAN.

6.3.3. Measuring system rate of recovery from incorrect predictions

When an incorrect prediction is made, the PIRR system experiences a spike in IL. After that incorrect prediction, the following few predictions will be delayed, too: it takes time for the PIRR system to recover from that initial incorrect prediction. Each incorrect prediction will "reset" the length of the "recovery period".

The time it takes for a PIRR system to recover from incorrect predictions (the "recovery period") is defined as the difference in time between the start of an increase in IL due to incorrect predictions, and the point at which the effect of those incorrect predictions have worn off and the system IL has returned to its normal IL operating level. Measuring recovery times may be important in building an understanding of how incorrect predictions impact IL, and are likely to be an important consideration when choosing and evaluating a prediction model.

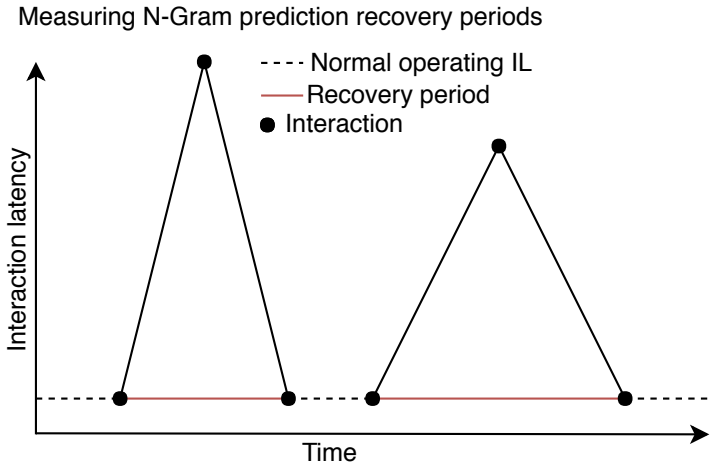


Figure 6.15 N-Gram prediction recovery periods.

When an incorrect prediction occurs, a spike in latency is observed and after some period of time, latency returns to its normal operating level as correct predictions are made. The recovery period (illustrated in Figure 6.15) is an important metric as it enables one to determine how quickly a prediction model (and indeed the system as a whole) recovers from incorrect predictions. When inspecting the raw data (see Figure 6.7, for example), it was noticed that the system recovery times appeared to increase with Order, as well as with MPA. To test this, the recovery times for all N-Gram Order and MPA experiments were calculated. Each experiment run produces an output file with IL measurements and associated timestamps. Each N-Gram Order and MPA experiment is repeated 5 times and therefore each experiment has 5 associated measurement files. Each of these files are analysed, their average recovery time is calculated and then the mean recovery time for those 5 files is determined. To calculate the recovery time for a single measurement file, the following steps are required:

1. Find all local maxima indices (positions of the peaks).
2. For each peak, find adjacent peaks (one on the left and one on the right) so that you now have three peaks (left, middle, right).
3. For each triplet of peaks, get all measurements between the middle peak and the left peak (il_left) and between the middle peak and right peak (il_right).
4. For each il_left and il_right, calculate their means: $\overline{IL_{left}}$, $\overline{IL_{right}}$. These means inform us what the "normal" IL is.

5. Iterate over all values in il_left and record the index *startIndex* of the first value found to be equal to or less than $\overline{IL_{left}}$.

6. Iterate over all values in il_right and record the index *endIndex* of the first value found to be equal to or less than $\overline{IL_{right}}$.

7. Extract corresponding timestamp for the start and end index.

8. Subtract end time from start time to calculate recovery period.

9. Average all recovery periods for experiment.

Algorithm 3 Latency simulation

```
1:  procedure GETSTARTANDENDINDICIES(int[] ilMeasurements)
2:      results = tuple(int, int)[]
3:      peakIndicies = getIndiciesOfPeaks(ilMeasurements)
4:      for (i = 0; i < length(peakIndicies); i++) do:
5:          P_index = peakIndicies[i];
6:          tuple(int, value) [] IL_left = ILBetweenPeaks(ilMeasurements, P_index,P_index − 1)
7:          mean(IL_left) = mean of IL_left
8:          tuple(int, value) [] IL_right = ILBetweenPeaks(ilMeasurements, P_index,P_index + 1)
9:          mean(IL_right) = mean of IL_right
10:         for (IL_tup in IL_left) do:
11:             if IL_tup.value > mean(IL_left) then:
12:                 startIndex = IL_tup.index-1
13:             else startIndex = P_index−1 + IL_tup.index
14:         for (IL_tup in IL_right) do: endIndex = P_index + IL_tup.index
15:             if a IL_tup.value > mean(IL_right) then:
16:                 endIndex = IL_tup.index+1
          return results.Add(tuple(startIndex, endIndex))
```

Once the start and end indices for a set of IL measurements has been found, they can be used to extract the corresponding start and end times from the set of IL measurement and recovery periods can then be calculated. The code for this can be found in A

To test this, the simulator was run first with the various latencies described in Table 6.1 and N-Gram orders 1 to 5, resulting in 150 experiment runs. The results of those experiments are shown in Figure 6.16, where it is clear that increasing N-Gram Order does impact the PIRR simulator given that in all simulated latencies, recovery times increased with Order.

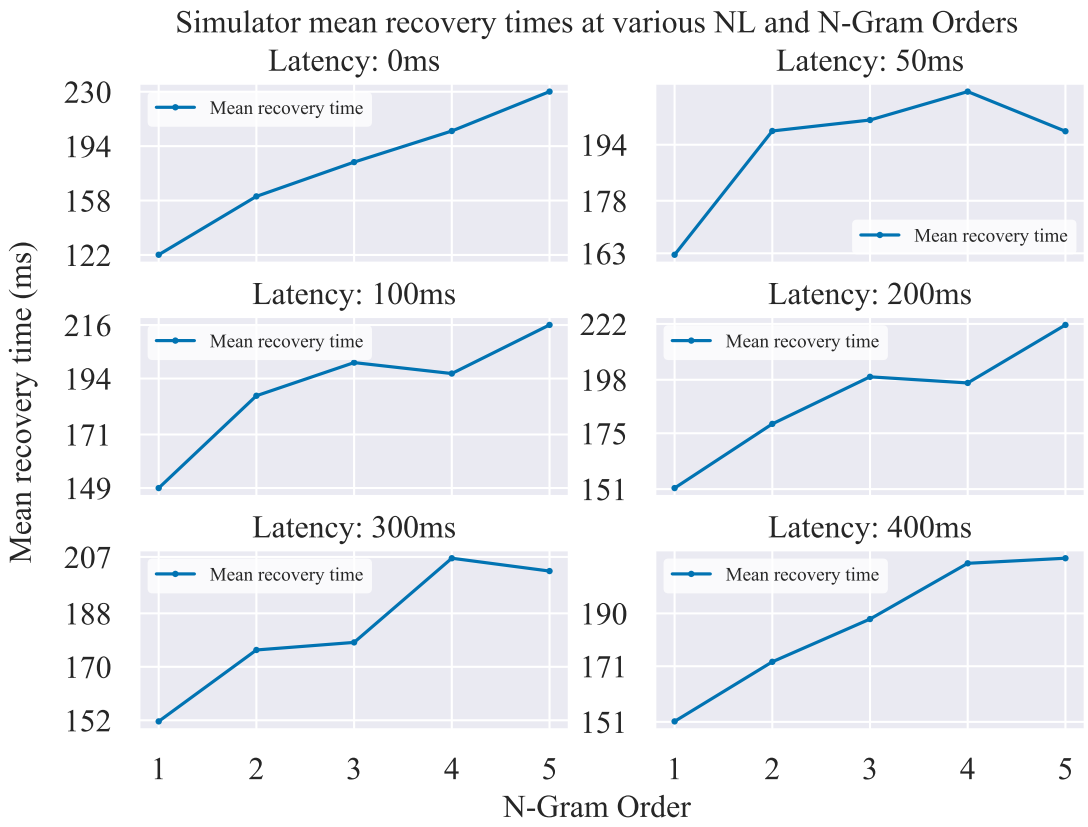


Figure 6.16 Simulator recovery times measured at various simulated NL values

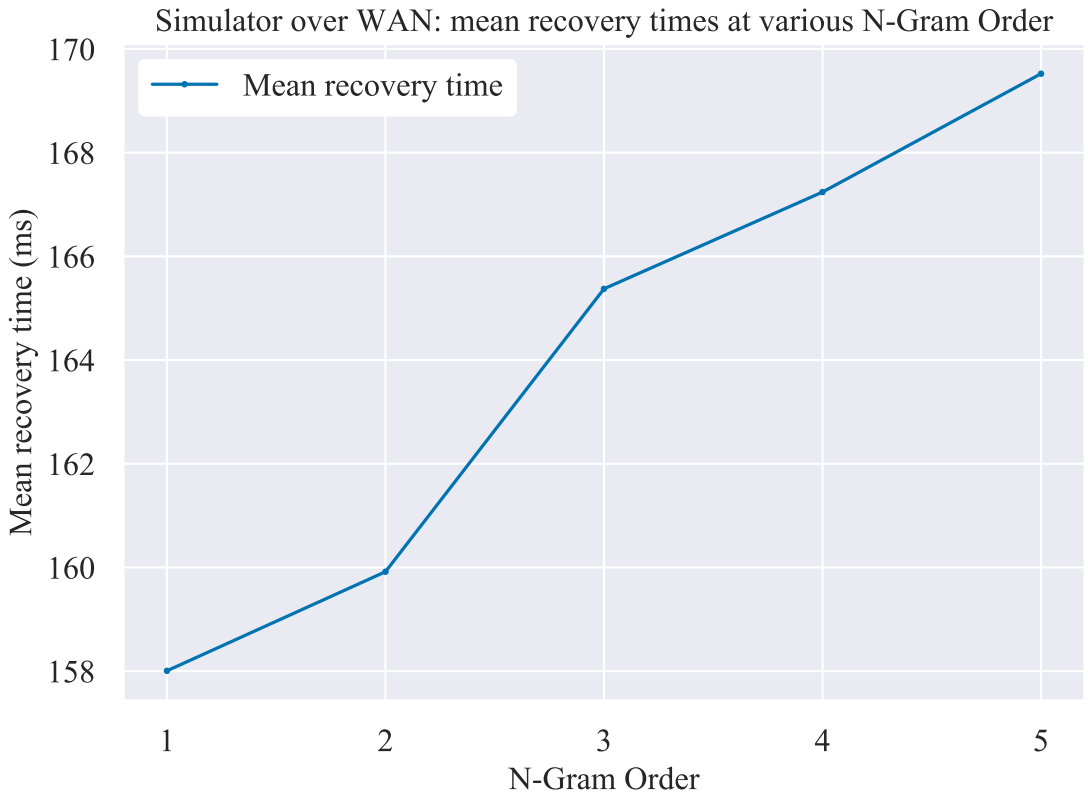


Figure 6.17 Simulator recovery times over WAN at various N-Gram Orders.

The same set of experiments were then performed over WAN, without any simulated NL. Figure 6.17 presents the results of the experiments run over WAN. Each point represents the mean

recovery duration for a single experiment repeated five times at a different N-Gram Order. As can be seen, the results are similar to when using simulated NL: the amount of time the system takes to recover from an incorrect prediction increases with Order.

Next, the recovery times of the PIRR simulator were calculated with respect to MPA. 300 simulation runs were performed: a simulation at each latency described in Table 6.1, for MPA's 1 to 10 and each experiment was repeated 5 times and averaged. Similar to the experiments performed against N-Gram Order, Figure 6.19 shows that mean recovery periods also increases with each increase in MPA and performing the experiments over WAN shows similar results.

The reason for the increase in mean recovery time with respect to both N-Gram Order and MPA is that while both can have a positive impact IL, they can also negatively impact IL - this was shown in 6.3.1 and 6.3.2. However, upon closer inspection, it can be seen that the effect of MPA on mean recovery times is greater than that of N-Gram Order. This is because while N-Gram Order impacts the likelihood of an incorrect prediction, MPA increase the chance of additional errors following an incorrect prediction. The larger the MPA, the more mispredictions will occur and as a result, the time the PIRR system takes to recover will increase, too.

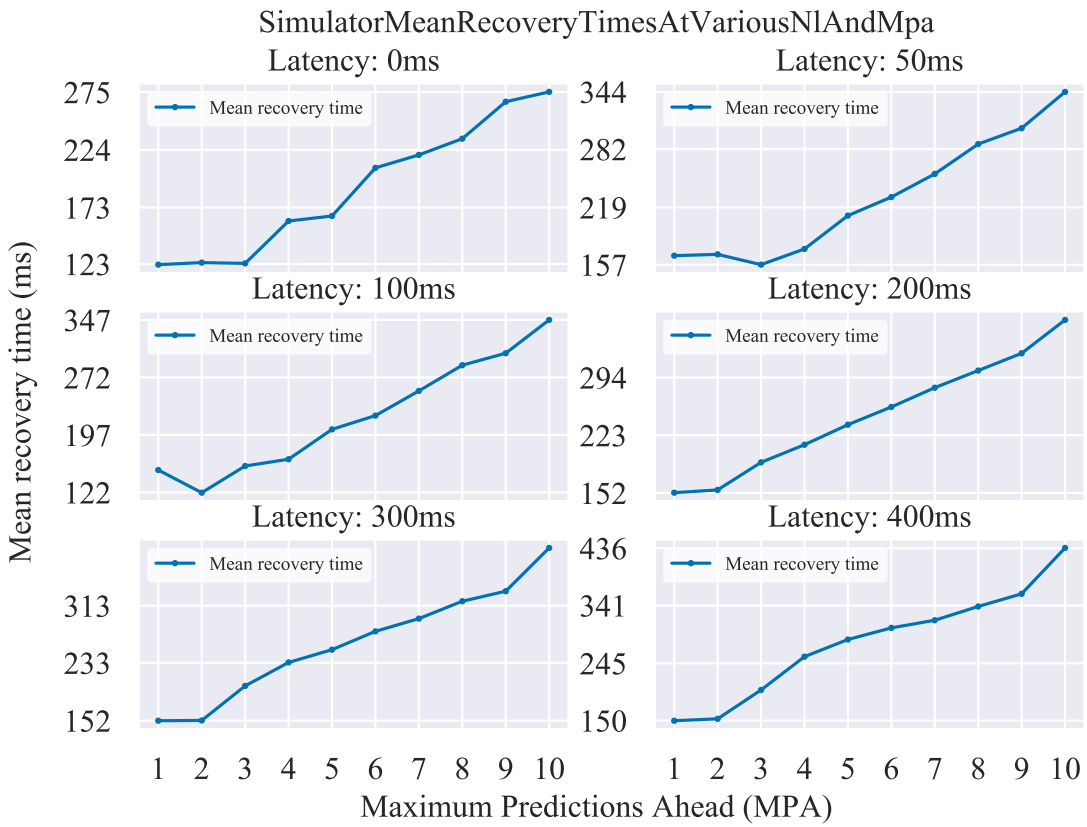


Figure 6.18 Recovery times of simulator at various NL and MPA.

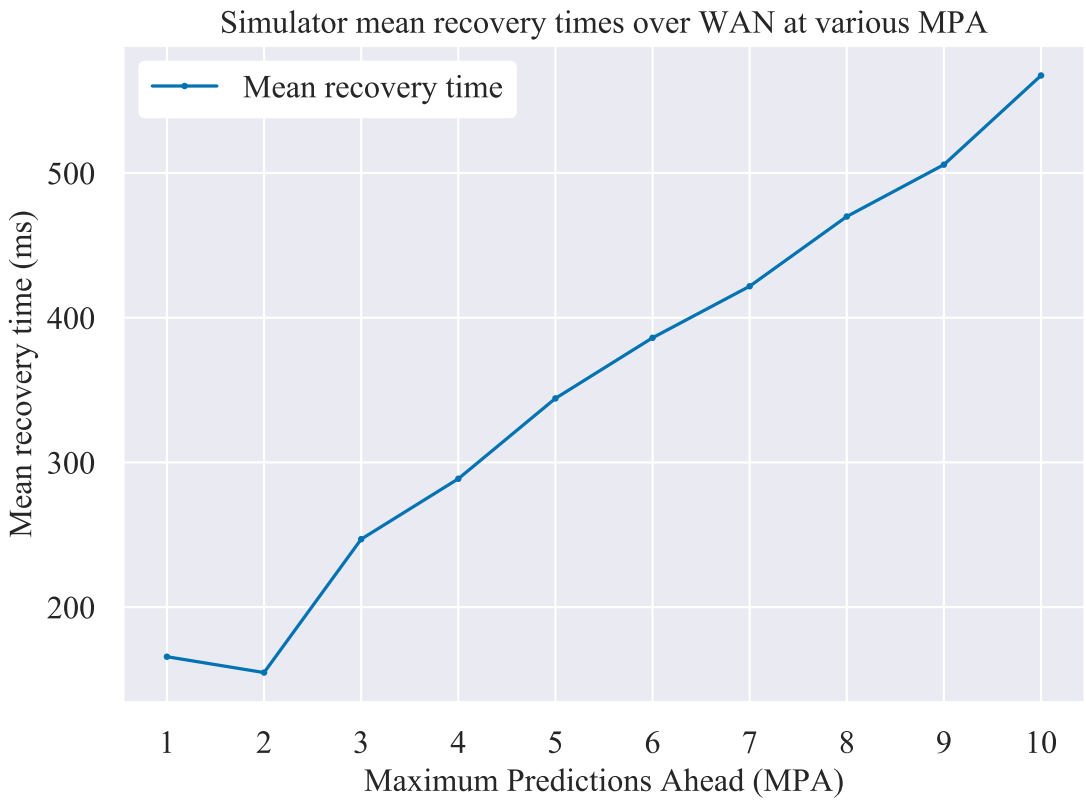


Figure 6.19 Recovery times of simulator running over WAN at various MPA.

6.4. Summary

In this chapter, the PIRR simulator described in Chapter 5 was used to examine the effect of prediction on IL, as well as the effect of N-Gram Order and MPA on IL. In addition, it was described how the recovery period of a PIRR system can be calculated, and the effect of N-Gram Order and MPA has on it.
First, base IL measurements were taken and the simulator was found to be operating as expected, given the 30ms simulated render delay and the introduced simulated latencies. The effect of the prediction module on IL was then examined in §6.3, where it became clear that the it is able to lower IL substantially, providing the client application with future results before the corresponding actions have been performed.

IL was found to increase with N-Gram Order, which after further investigation, it was revealed that the reason for this increase is due to an increased number of incorrect predictions. The effect of MPA on IL was inspected next and it was found that as NL increases, a greater number of predictions must be made in order to mask the introduced IL. By increasing MPA, higher NL's can be masked. However, when MPA increases, IL decreases only until a threshold is reached, at which point IL begins to climb. How far ahead the prediction module must predict is depended upon the total RTT and the rate at which interactions occur on the client application (SD). A method for determining the ideal MPA has been introduced and described.

In addition, a method for calculating system recovery times from incorrect predictions, with respect to both N-Gram Order and MPA, was introduced. It was shown that for both N-Gram Order and MPA, mean recovery times increase regardless of simulated NL or whether using a WAN.
It was also noticed that MPA appears to have more of a negative impact on recovery times than N-Gram Order. This is likely due to the fact that while N-Gram Order increases the chances of a misprediction, MPA causes the number of mispredictions to increase proportionally to the value of MPA, after a misprediction.

It can be concluded that the prediction module can play an important role in reducing IL in IRR systems and that attention must be given to MPA so as to ensure a backlog does not form on the client. While an N-Gram model was used here, another model might offer better prediction of user interactions, however the effect of MPA should remain the same. Finally, using the method described to calculate recovery times, the system could be monitored so as to further improve PIRR performance; for example, by adjusting the MPA based on sampled network delay.

Chapter 7. Exploring the effect of N-Gram prediction on interaction latency using a real-world PIRR system

7.1. Overview

Until now and except for when experiments were performed over WAN, all delays have been simulated using the latency simulator described in §3.3. The PIRR simulator provided a controllable environment where all aspects of the system could be controlled, measured and evaluated. While the simulator platform was used to explore the effect of N-Gram Order and MPA on IL, it is unclear whether those the experiments performed and results gathered are representative of a real-world PIRR system. Therefore, in this chapter, the effect of N-Gram Order and MPA are explored using a purpose-built PIRR system.

Towards this, this chapter introduces a real-world PIRR system design using Unity3D, a popular 3D games engine. Using this platform, we investigate the effect of prediction, N-Gram Order and MPA on IL. Results gathered through experimentation with platform are compared with that of the simulator. This is important if the simulator platform is to be used as a model for experimenting with and introducing modifications, before changes are deployed to a live system. In addition, we describe problems encountered with user action management and interaction with scene objects, and present approaches to mitigate these issues.

7.2. Architecture

This section describes the architecture of a PIRR system built with Unity3D and modelled on the simulator. Both the client and server applications were built with Unity3D and communications were managed exactly as they are described in §5.3, using RabbitMQ. All architecture definitions are the same as stated in §5.2. An initialization message is also used and is identical to that described in §5.3.1. User interactions were collected from 5 randomly generated and navigated 3D mazes (see §4.5) and were used for all experiments in this chapter. Finally, the prediction module is identical to that described in §5.3.3.

7.2.1. Client application

The client scene consists of a primitive plane object with a camera placed directly above it such that when in “Play” mode, only the plane is visible. The camera-plane setup is the same as that described in §3.6.1 and illustrated in Figure 3.4. There are no other scene objects. Besides reading interactions, transmitting them to the prediction module, receiving frame result

messages from the server application and displaying the received image, the client application is also responsible for determining whether interactions are valid (i.e. whether the performed interaction does not result in an illegal move such as one which would allow the user to move through a wall) – but this is only done when operating in manual mode.

It is relatively straight forward to map interactions into Unity3D. Since Unity3D is a games engine, it has utility functions for detecting user input built into its API. For example, Input.GetKey("A") will return True if the user has pressed the "A" key. Unfortunately, Unity3D does not provide a way to feed interactions into its input system. In other words, there is no mechanism to simulate or create interactions. While this is typical of most game engines, it presents an issue when automating IRR experiments.

One approach is to simulate interactions. This can be achieved by hooking into the underlying OS event stream using the User32 library (Windows only) or by using WindowsInputSimulator, a popular open-source project. Unity's input system should detect these simulated interactions and process them as if they were physically performed. Unfortunately, during system testing, it was found that at times simulated keyboard events were not registered by Unity3D's input system and mouse input was not registered at all. For this reason, interaction is not simulated in the traditional sense where user events are raised at the OS level. Instead, interactions are read from the same interaction template described in §5.3.

When the client application starts up, it reads in a configurations file and sets various properties such as experiment parameters and the amount of latency to introduce. As with the simulator, an initialization message is sent through the system, causing the client application to wait for confirmation that all system components are ready. When the system is ready, it may begin processing the interaction queue and results arriving in the CFB.

Unity3D has two useful overridable functions called Update and FixedUpdate. Update executes once per frame but unfortunately does not provide a way to control exactly when it is called. On the other hand, the execution of FixedUpdate may be controlled by setting the Fixed Timestep in the Unity3D Editor and was therefore chosen as the function of choice to process the interaction queue. This function is executed at some specified interval, is managed entirely by Unity3D and therefore does not require a separate task to be created and started (as explained in §5.3.1) for the sending of interaction messages to the prediction module. On each iteration, an interaction is read from the interaction queue and a new position is calculated given the interaction. If running in manual mode, the new position is checked to ensure that the action will not result in the player avatar going out of bounds; this is done before sending the interaction to the prediction module and therefore prevents the introduction of invalid actions to the system. However, when operating in automatic mode, this step is skipped since the interaction template contains no invalid actions. A new M_c is then created and the action, number and position properties are set.

M_c is then sent to the prediction module, a $M_{pending}$ is created, the stopwatch is started and finally, $M_{pending}$ is added to a pendingInteractions queue (identical to the one used by the simulator).

The Receive task, like the one used in the simulator, loops until system shutdown. When a M_s arrives, it is passed to the latency module where it is delayed for a specific amount of time. When the delay has expired, the M_s is passed back to the client application and is added to the CFB.

When using the simulator, results arriving from the server application are not displayed because a simple dummy frame is delivered, and rendering is simulated. However, in this live system, real render results are transmitted from the server application to the client and therefore need to be displayed to the user as real-time visual feedback to any interactions performed. Unfortunately, Unity3D is single-threaded. That is, the engine does not support multi-threading and is not thread-safe. As a result, all received results must be passed to the main Unity3D thread so that the render result can be displayed. To display the frame to the user, the byte data from the received $M_{s.Frame}$ is converted into a Texture2D and applied to the surface of the Plane. This happens before the Update function is called so that the delay between receiving a result, writing it to a Texture2D and having it visible to the user is kept at a minimum.

7.2.2. Prediction module

he prediction module is the same described in §5.3.3; they are identical.

7.2.3. Server application

The Unity3D server application differs from the simulator server application in that it performs real rendering and transmits actual frame results to the user which are presented by the Unity3D client application. As a result of real rendering, control over render delay is lost and is at the mercy of the Unity3D Rendering Engine. Another notable difference is that user interactions must be processed, mapped to the scene and transformations to the "player" must be made. Further, the user is able to interact with scene objects. Nevertheless, the server application behaves in much the same way as that of the simulator version and this section will therefore describe the overall operation, while §7.3 will describe more broadly the issues that arise from using prediction in an IRR system and how those difficulties were overcome.

When an interaction message (M_p) arrives on the server application from the prediction module, it is put into a queue where it waits until the next render loop. If a message is available, it is dequeued and the predicted (or non-predicted) corresponding action is processed. Processing the action involves updating a player (a 3D sphere, discussed in the next section) position. The game output frame is then captured by reading pixels from the frame buffer into a Unity3D "RenderTexture". The data from RenderTexture is then encoded to a JPEG using a 70%

compression ratio (Unity3D default). Finally, a message is created and transmitted to the client application.

7.3. Interacting with scene objects

As mentioned, a fundamental difference between the simulator and the Unity3D server applications is that real interaction and rendering is taking place. Also mentioned is that the introduction of the prediction module results in some challenging situations. These issues were observed while developing the PIRR system and were not anticipated during the development of the simulator. As a result, the two platforms underwent simultaneous periodic changes, and can be considered to have been developed concurrently. In the experiments performed using the simulator, all interactions were read from a template using the automatic mode. Since there was no scene, there was therefore no need for interaction between the player and the scene objects. In a real-world PIRR system, however, the user steers the simulation and must be able to interact with the scene. For visual reference during development, a simple 3D maze scene was developed: various walls form passages are navigable by a "player" (a 3D sphere) and 10 collectable "coins" scattered around the scene.

The first issue discovered was that while the simulator is simple in that all interactions are performed automatically; in this system the user must have control. In a typical video game or interactive visualization, the user must perform interactions on his or her local device. Those interactions are then detected by the visualization software, processed and a frame displayed. Since the server application must run remotely and the user therefore does not perform interactions on the same machine as the rendering application, interactions must be managed appropriately. Unfortunately, Unity3D does not allow interactions to be fed into its input system, which means that the Input controller exposed by Unity's API is of no use. Further, simulating interactions does not appear to be reliable since during development, it was observed a number of times that simulated interactions were no detected by Unit3D using this technique would therefore mean that there could be no guarantee that repeated simulation runs were identical. Therefore, when an interaction arrives on the server application, it is processed, and the scene is updated directly, without being simulated. Unfortunately, updating the scene directly causes another problem and thus does not provide a simple solution. Specifically, the physics engine built into Unity3D allows for player movement, collisions with game objects, the simulation of gravity, etc.

By default, physics is enabled, which means that it is possible that an interaction may occur over multiple frames and would therefore cause a serious issue for measuring IL: the initiation point of an interaction may be known, but how would the system know when the interaction has stopped: when the user stops performing an interaction, when the player avatar stops moving? What if the player avatar moves (i.e. sways) without user interaction? To counter this, physics was disabled on the server application. Having physics disabled and interactions applied directly

to the scene makes the server application state-like. For example, when an interaction such as "W" (representing a forward movement) arrives, the position of the player avatar can be updated from 0, 0, 0 to 0, 1, 0. While this is limiting in that interactions are made to be discrete, it does allow for exact measurements of IL because there is no ambiguity about whether or not a frame result directly corresponds to a specific action. Additionally, this makes prediction easier as it is possible to identify whether the predicted location exactly matches the actual one that results from a user action and therefore also makes it easier to identify incorrect predictions. Unfortunately, disabling physics results yet another issue where collision detection ceases to function: the player avatar will no longer collide with walls and or other scene objects, and instead will continue through previously impassable objects.

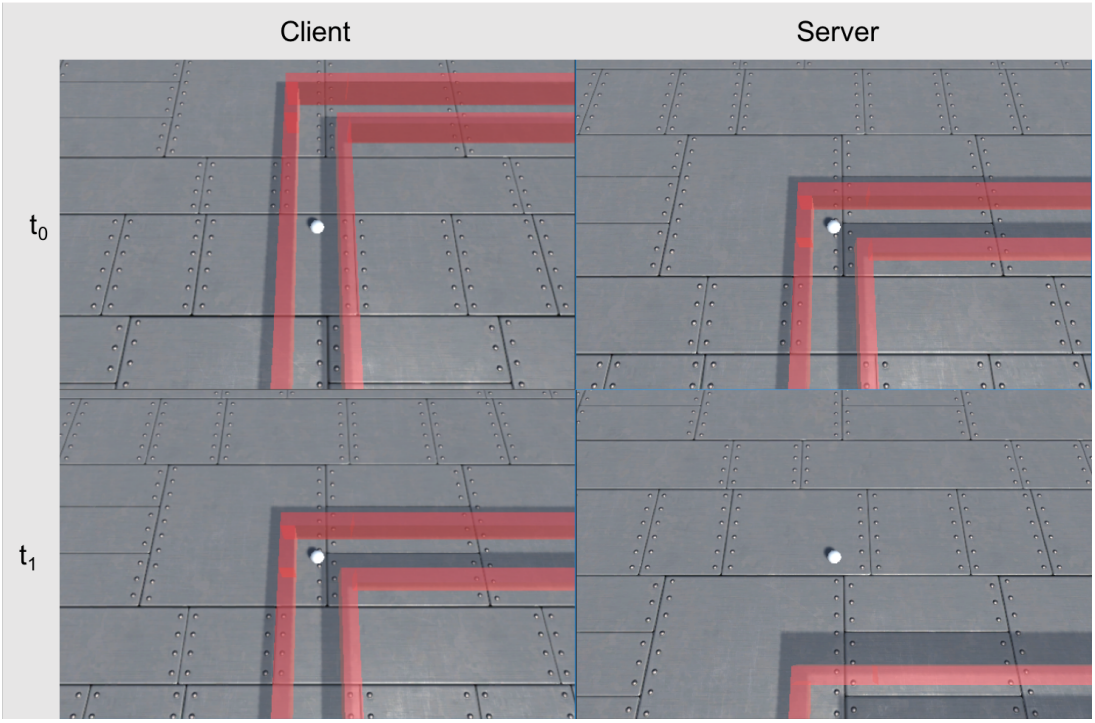


Figure 7.1 Client application vs server application views.

Figure 7.1 shows the view on the server and on the client at time t_0 and t_1. At t_0 the player has moved forward and is approaching the top wall. Due to prediction, the server application is executing ahead of the client application and therefore shows a position closer to the top wall. At t_1 the player has moved closer to the wall and can now only move forward 1 place more. However, the server application passes through the wall. The client application will never present those invalid interactions to the user because of the validation check performed. On the server application, a check is performed to determine whether the player has collided with a coin. If it has, the coin is removed from the scene and the updated frame is transmitted to the client.

7.4. Experiments

7.4.1. Base IL

The measurement of the base IL of the system is useful for determining any irregularities or abnormal system behaviour and when compared with the same experiment from the simulator, should inform whether or not the two systems are comparable, at least in terms of baseline performance. Therefore, the same set of experiments described in §6.2.1 were conducted (no prediction, executed 5 times and the results averaged) and identical parameters to those experiments were used, with the exception that the rendering delay was not simulated and the resulting frames are displayed on the client. By performing live rendering, it is not possible to control the rendering delay parameter. Therefore, when comparing base IL across various introduced NL, it impossible to differentiate processing delay from render delay as presented in Figure 6.2. Nevertheless, in order to keep render delay as close to that of the simulator, vsynch was enabled and the "Application.targetFrameRate" property of the Unity3D server application was set to 30. Figure 7.2 presents the base IL measured in the live PIRR system, run locally and without any injected NL.

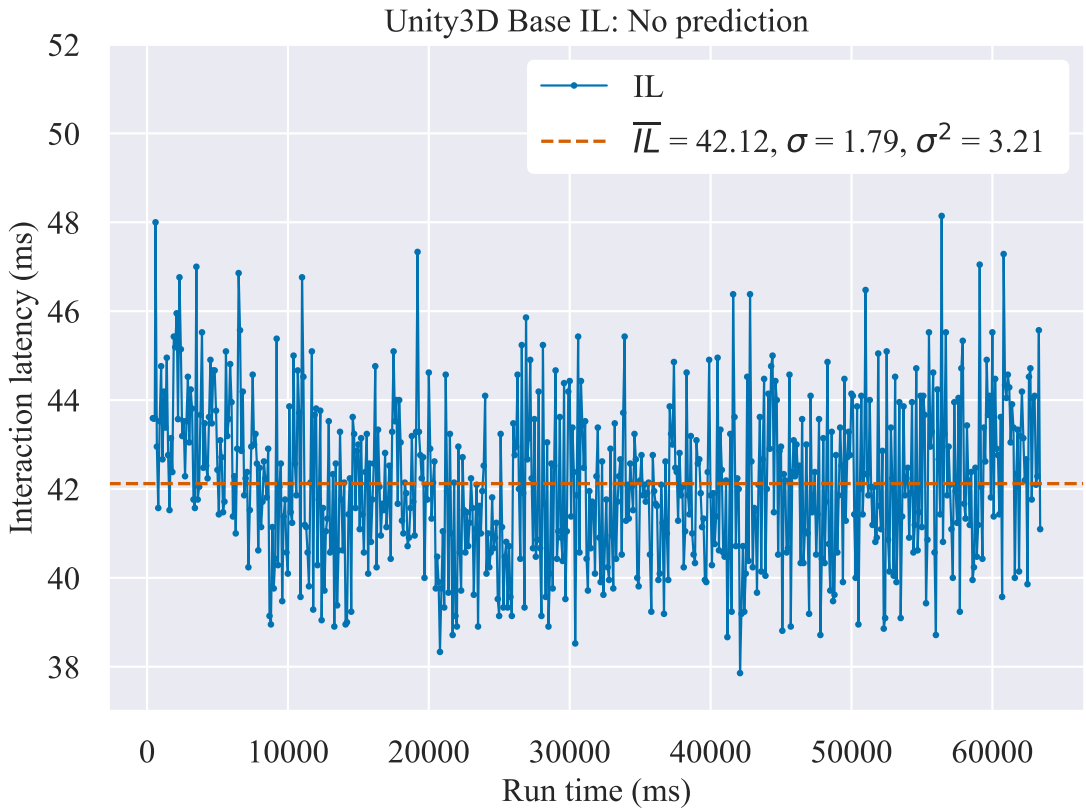
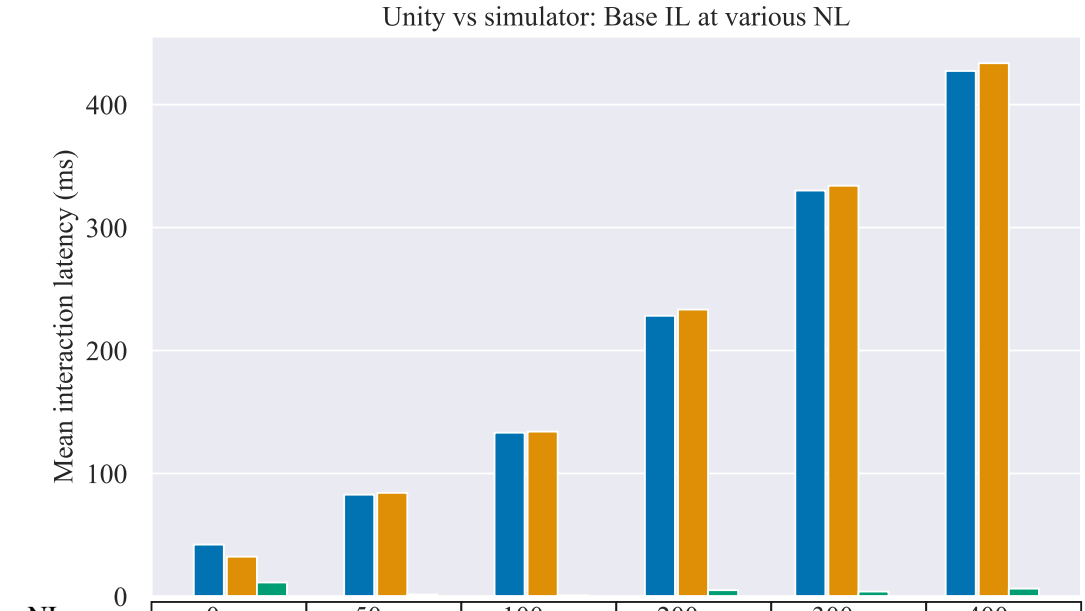


Figure 7.2 Baseline IL measurements collected from Unity IRR system.

The Unity3D system was then run with the six latencies from Table 6.1 (0, 50, 100, 200, 300 and 400ms). Like with the simulator, experiments were repeated 5 times and their results were averaged. In Figure 7.3, the results from these experiments are compared with those from the simulator. From the figure, it can be seen that the simulator and Unity3D PIRR system behave comparatively with a mean delta of 4.83ms between the two platforms. The Unity3D PIRR

system was then run over WAN, with each experiment being repeated 5 times, three times a day. Figure 7.4 compares the results of these experiments with those of the simulator using WAN (see Figure 6.8).



NL	0ms	50ms	100ms	200ms	300ms	400ms
Unity	42.12	82.73	133.14	228.25	330.12	427.39
Simulator	32.28	84.21	134.03	233.3	334.04	433.7
delta	11.31	1.48	0.89	5.05	3.92	6.31

Figure 7.3 Unity3D vs Simulator base Interaction Latencies.

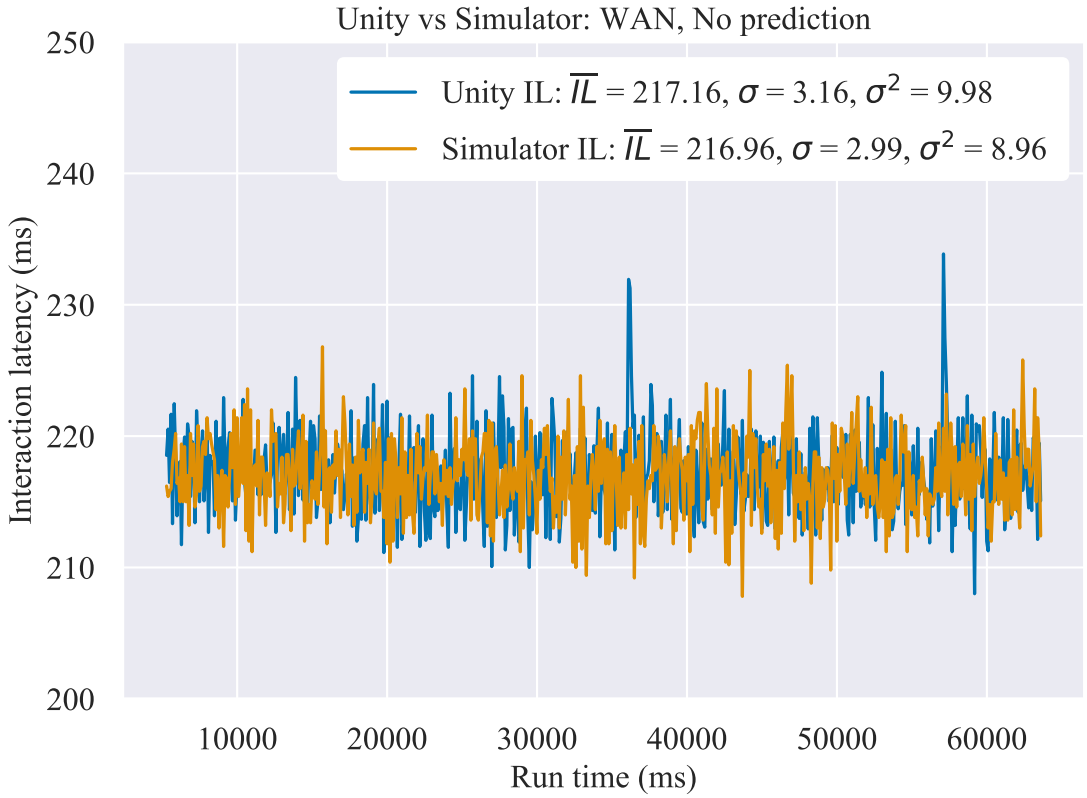


Figure 7.4 Unity3D Interactive Remote Rendering system vs Simulator over Internet without prediction].

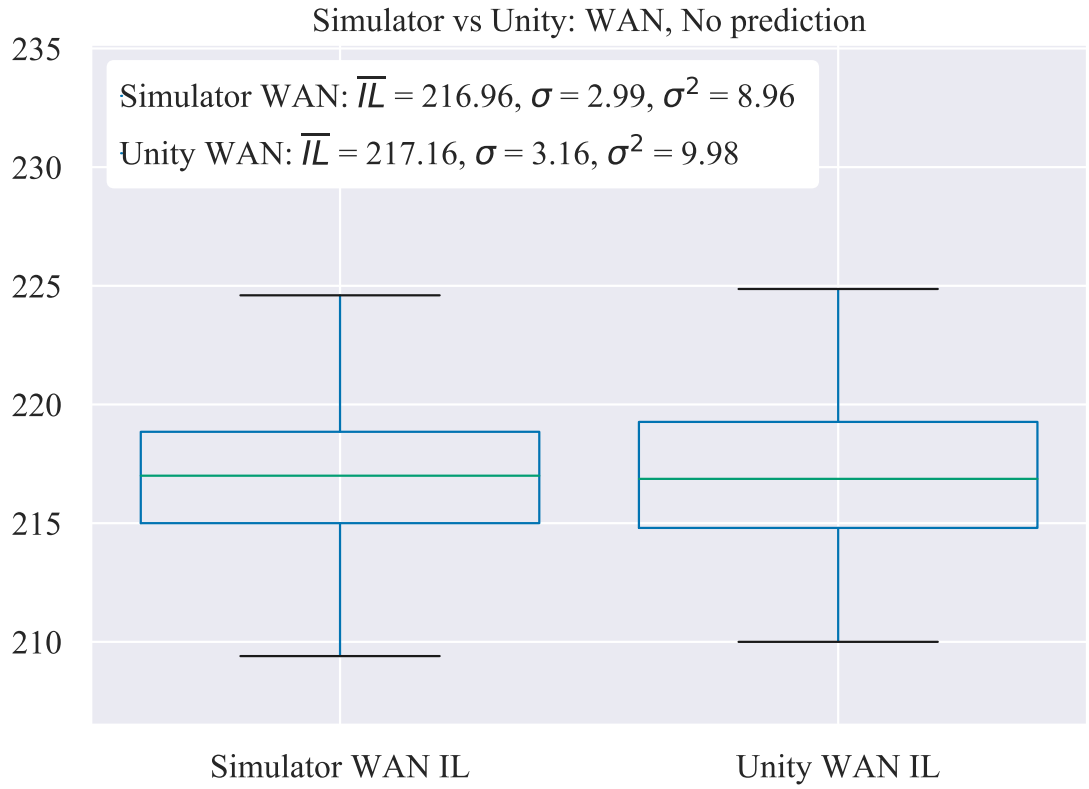
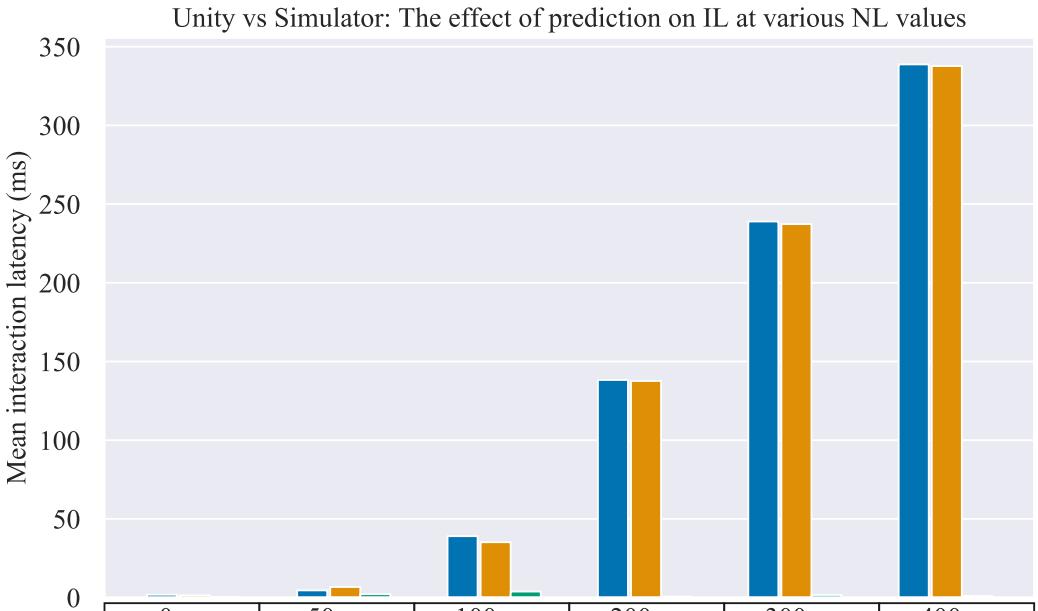


Figure 7.5 Unity3D Interactive Remote Rendering system vs Simulator over Internet without prediction. Box plots

From Figure 7.4 and Figure 7.5, it can be seen that the Unity3D system operates well and its IL measurements correspond well with those of the simulator when running over WAN. The Unity3D system IL varies more than the simulator, but this is likely a combination of various factors such as rendering being controlled by Unity3D and is therefore out of our control, as well as the use of Thread.Sleep and the difficulties of performing short delays in an non-real-time OS, such as Windows.

7.4.2. The effect of user interaction prediction on IL using the IRR system

As with using prediction during the simulator experiments, the hypothesis is that employing the prediction module will have a positive effect on the IL of the Unity3D PIRR system. The same set of experiments conducted in §6.2 (see Figure 6.6) were performed here, and Figure 7.6 compares the results collected from the Unity3D system with those from the simulator.



NL	0ms	50ms	100ms	200ms	300ms	400ms
Simulator	1.82	4.57	39.02	138.26	238.95	338.75
Unity	1.4	6.73	35.18	137.62	237.29	337.71
delta	0.42	2.16	3.84	0.64	1.66	1.04

Figure 7.6 Effect of prediction on Interaction Latency at various simulated Network Latencies: Unity3D Interactive Remote Rendering system vs Simulator

As can be seen in Figure 7.6, the simulator and Unity3D measurement values are very similar across all simulated NL values. The difference between the two platforms (delta) reaches a high of 3.84ms. The simulator averages slightly above Unity3D. However, IL for the Unity3D system does vary more than that of the simulator, as can be seen in Figure 7.7 and represented with boxplots. Figure 7.8 compares raw mean IL (the average of 5 experiment runs) for the Unity3D system with the simulator running with 50ms simulated NL, MPA = 1 and N-Gram *Order* = 1. Similar to Figure 6.7 in §6.3, IL is considerably lower using prediction (although large spikes of IL are present, indicating incorrect predictions). In this plot it is clear that Unity3D measurements are more variable than those of the simulator.

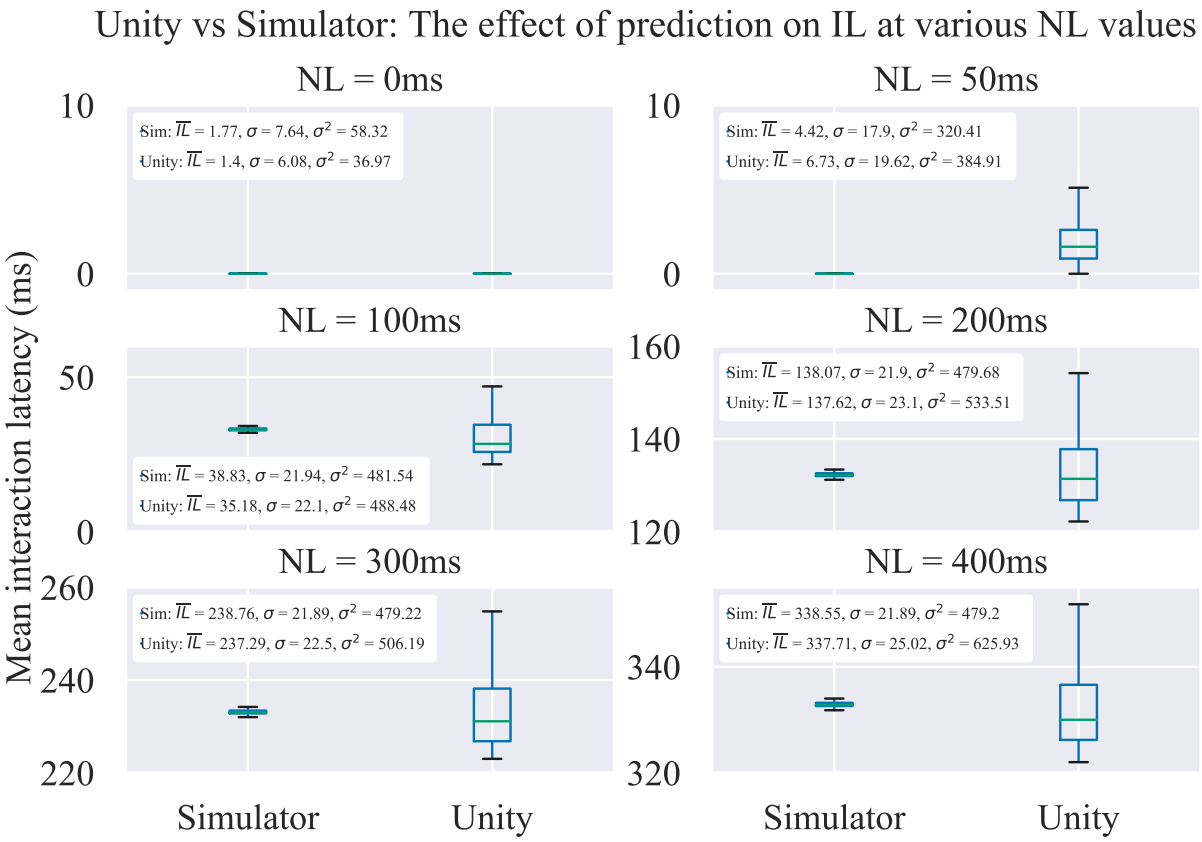


Figure 7.7 Box plots comparing the Simulator with Unity3D system at various Network Latency values.

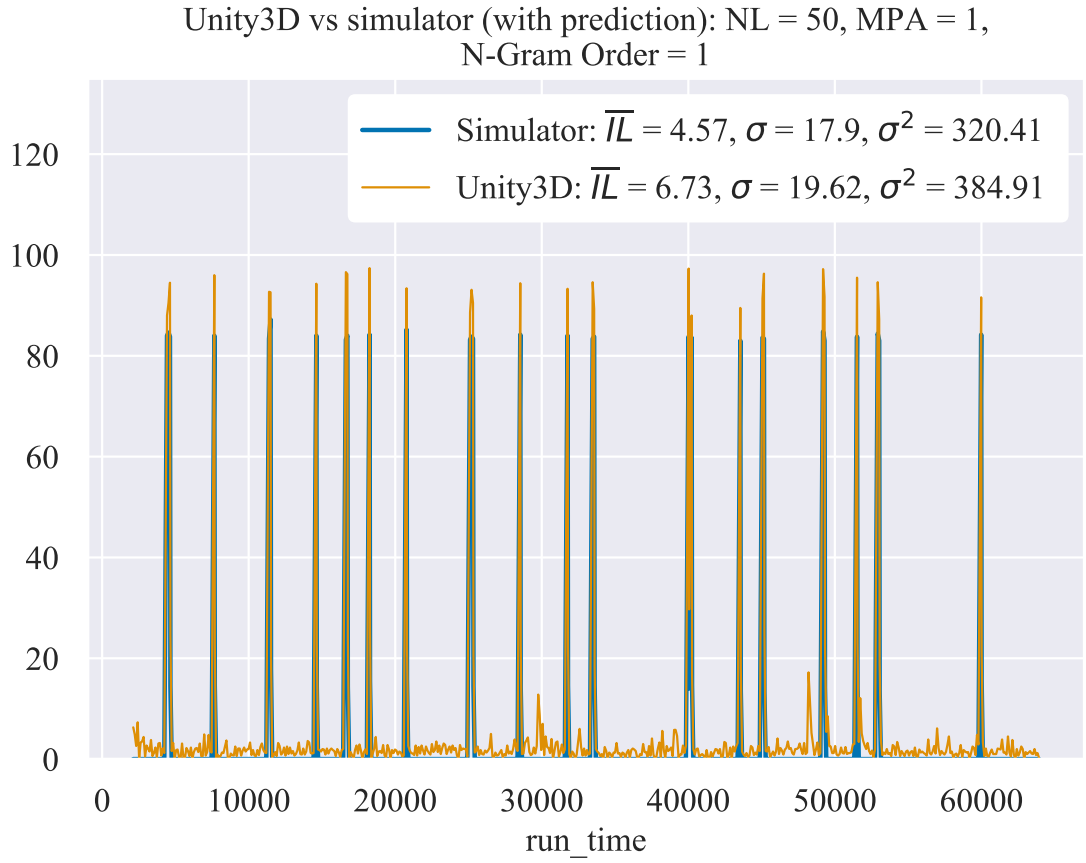


Figure 7.8 Unity3D Interactive Remote Rendering system vs Simulator, with 50m simulated Network Latency.

7.4.3. *The effect of N-Gram Order on IL*

In §6.3.1, the effects of N-Gram Order on IL were examined using the simulator. This section explores the same set of experiments, but with the real-world PIRR system. As a reminder, 6 latencies are used (0ms, 50ms, 100ms, 200ms, 300ms and 400ms), N-Gram Orders from 1 to 5 are evaluated and in all the following experiments, MPA is set to 1. After performing each N-Gram Order/NL experiment 5 times (resulting in a total of 150 experiment runs), results were averaged, and the results were compiled and are presented below in Figure 7.9.

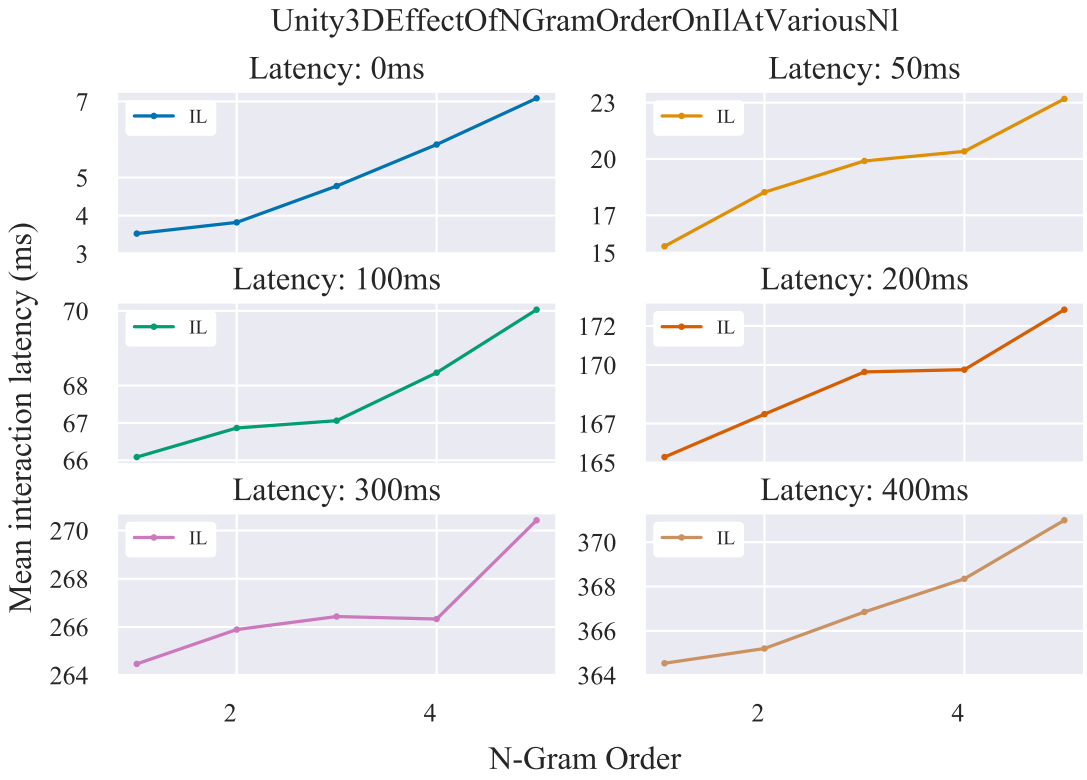


Figure 7.9 Unity3D Interactive Remote Rendering system: effect of N-Gram Order on Interaction Latency at various Network Latency values.

From Figure 7.9 it can be seen that for all simulated latencies, mean IL values increase with N-Gram Order and the increase in mean IL is between 1 and 2ms per N-Gram Order increase. Next, the system was run over a WAN for each of the N-Gram orders and again with MPA set to 1. As can be seen in Figure 7.10, IL increases with N-Gram Order, indicating again that the system performs better (in terms of IL) with lower amounts of history used for creating predictions.

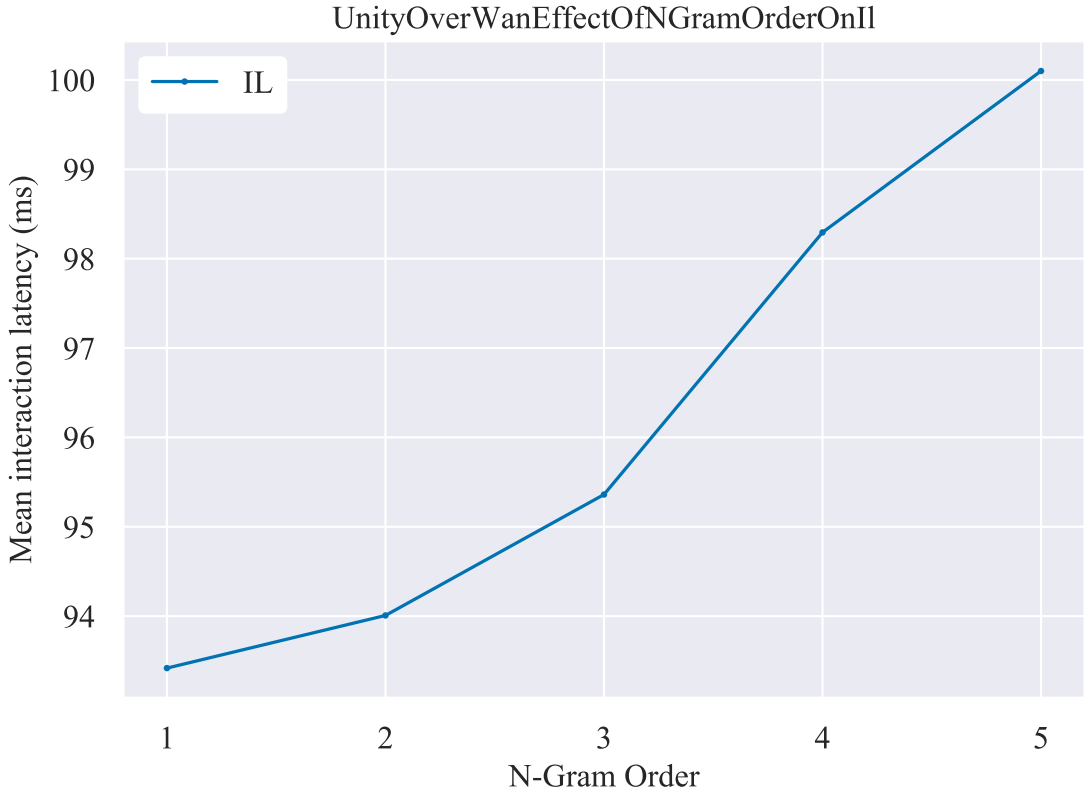


Figure 7.10 Unity3D Interactive Remote Rendering system: effect of N-Gram Order on Interaction Latency with 1-step ahead prediction

7.4.4. The effect of MPA on IL

The same set of experiments performed in §6.3.2 were repeated for the Unity3D system. As a reminder, the parameters used were: 30ms simulated render delay, MPA values from 1 to 10, N-Gram Order was set to 1 and simulated NL values of 0ms, 50ms, 100ms, 200ms, 300ms and 400ms. Each MPA/NL experiment was repeated 5 times and the results of those 5 experiment runs were averaged. Figure 7.11 illustrates the results from these experiments.

From Figure 7.11, it is clear that, like the simulator, MPA has a positive effect on the average IL of the system. While at lower NL values it is less obvious, the system reaches an "ideal" MPA which is dependent upon the introduced NL. For example, when NL = 400ms, the ideal MPA is 5 because IL is, on average at its lowest in comparison to other MPA values. Once the ideal MPA has been reached, increasing MPA further results in the mean IL of the system beginning to climb.

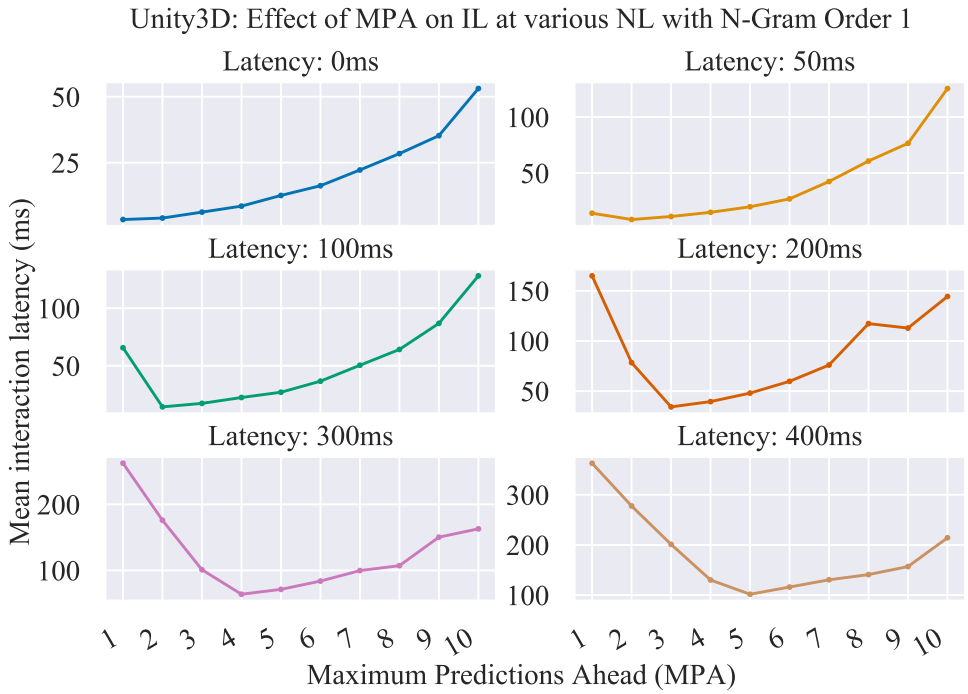


Figure 7.11 Unity3D Interactive Remote Rendering system: effect of Multiple Steps Ahead prediction on Interaction Latency at various simulated Network Latencies

Next, the system was tested over WAN, with the client application located in Cambridge, UK, and the server based in Northern California, USA, and hosted on an Amazon EC2 instance. The same parameters as above were used for this experiment, except for the fact that no NL was simulated.

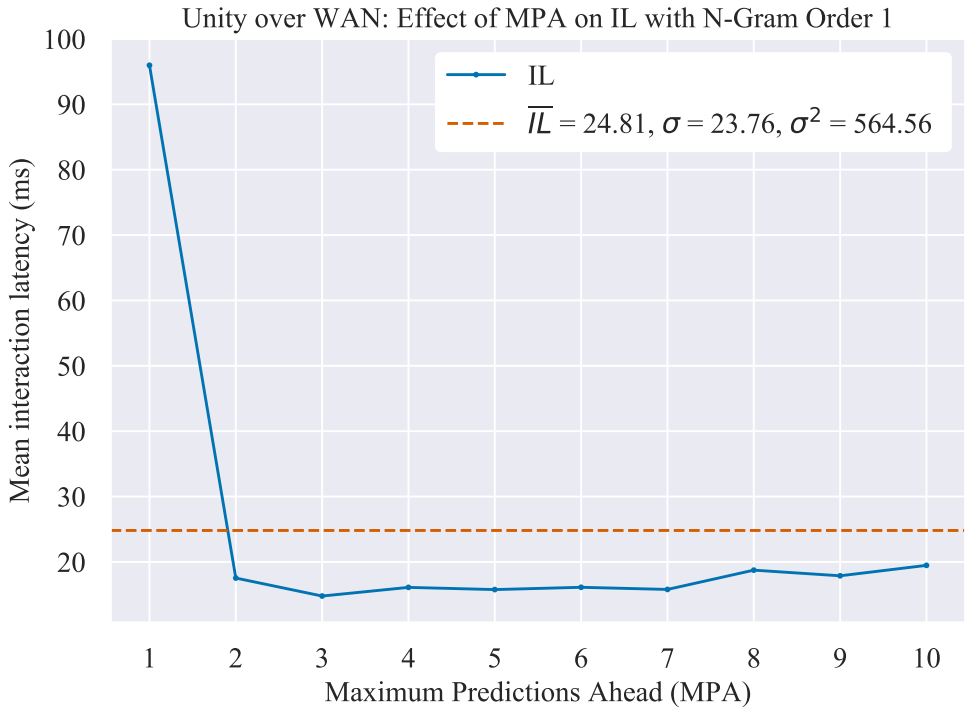


Figure 7.12 Unity3D Interactive Remote Rendering system over Internet: the effect of Multiple Predictions Ahead on Interaction Latency. N-Gram Order set to 1.

Figure 7.12 shows that just three predictions ahead is enough to minimise IL when the system is operated over a WAN. Interestingly, at MPA = 10, there is no sudden increase in mean IL (in comparison with Figure 7.11). However, this is likely due to the server application (the renderer) not running locally and therefore more resources are available to the client application, allowing it to process incoming messages more quickly.

7.5. Summary

This chapter explores the use of Unity3D as a PIRR system, how to interact with scene objects, how to integrate prediction and its effects on IL. The architecture of this system is similar to that of the simulator and both platforms consist of client, prediction and server applications. The fundamental differences between the architecture of this system and that of the simulator are the following: real rendering is used and results are output to the screen, which makes incorrect predictions visible to the user; real user interaction may be used and the user may therefore interact with objects in the scene; there is no control over the how long rendering takes since it is not simulated.

In a typical IRR system (without prediction), interaction is straightforward in that the user performs some action and simply waits for a response to be presented on-screen. However, the introduction of the prediction module led to some complications, specifically in determining how to match user actions to their correct and corresponding predicted frames, as well as how to store future prediction frames for when the system is operating ahead of the user. These issues were overcome by developing a pattern/tag incorporating information which allows future frames to be easily identified and stored for later use. In doing so, the issue of measuring IL was also solved, since both IL measurement and the user need action-frame matched pairs. Further, interacting with scene objects and experiencing situations such as object collision proved to be challenging, since the client has no knowledge of which actions are permitted. To remedy this, scene boundary information is transmitted from the server to the client, allowing the client to prevent the user from performing an illegal move, such as passing through a “wall”. A 3D maze with collectable “coins” was developed for the scheme to be tested on and is presented.

In terms of experimentation, N-Gram prediction was evaluated in the same way as done for the simulator. First, the base operating IL is presented, followed by a comparison of IL measurements between the simulator and the Unity3D system at various simulated NL values. The results show that the two systems operate comparatively, with just over 11ms difference when no simulated NL is used and an average of 4.83ms difference across all NL values. The effect of N-Gram Order was also considered, and results confirm that the system favours lower Order N-Grams, regardless of whether or not simulated NL or WAN is used. Finally, the effect of MPA on IL was also looked at and the results confirm that MPA has a positive effect on IL, but only to a certain value and again, regardless of whether or not simulated NL or WAN is employed.

Chapter 8. Managing incorrect predictions

8.1. Overview

In any system involving prediction, occasionally, a prediction will be incorrect. In the case of IRR systems, this means that unless an incorrect prediction is managed, the user will experience the full RTT of the system, as if no prediction were used. Predictions can fail in two key locations: while the user is performing interactions (such as moving forward, rotating a data set, etc) or between interaction periods (such as when no interaction is being performed and then suddenly an interaction is initiated from rest). If prediction fails while the user is interacting with the system, the user will experience a disruption in whatever task they are attempting to complete. For example, if navigating a VE or rotating a large set of data, the user will find the screen appears to "freeze" when an incorrect prediction is experienced, making the application feel "jerky". On the other hand, if prediction fails at the start of a sequence of interactions (e.g. the user begins moving forward or starts to rotate a data set), a large IL delay will be experienced, making the application appear to be less responsive.

In this chapter, various approaches to mitigating incorrect predictions are discussed. One approach, the multi-track scheme, is introduced and its effect on IL is investigated. This approach was chosen in an attempt to reduce the impact of misprediction on IL by predicting not only multiple steps ahead, but by also predicting multiple alternate interactions. The idea is that if the first prediction is incorrect, the "next best guess" may be correct and can be used. However, we first describe some other approaches to managing incorrect predictions.

8.2. Approaches for managing incorrect prediction

8.2.1. Local rendering

In events where prediction has failed and the client has received an invalid message from the server application, the client could make use of local resources and generate frames until the IRR system has recovered and correct message begin being received. This would unfortunately mean that the client device would be required to have sufficient compute and storage resources available, limiting the suitability of this approach to certain devices, require access to potentially sensitive data (for rendering) and generally erode the benefits that IRR systems streaming images afford.

8.2.2. Image warping

Image Warping may allow for the masking of incorrect predictions: the last known "good" frame can be warped to a new perspective according to the current user input, resulting in a new frame representing a scene change due to some action. This synthesized frame could provide the illusion of responsiveness to the user, while the IRR system attempts to recover from the incorrect prediction. Any image warping algorithm will require processing and resources to be available on the client, and as of yet, it is unclear as to how much latency this approach can mask. Additionally, as mentioned in §2.3.2, image warping suffers from hole artefacts, as well as from the inability to consider moving foreground and background objects. Therefore, image warping may only be of benefit when data is static such as in certain data visualizations.

8.2.3. Prediction

Image warping has the unfortunate drawback of not being able to consider the motion of foreground and background scene objects, severely limiting its role in masking incorrect predictions. However, neural networks were recently demonstrated to be capable of generating next-frame in video prediction using an unsupervised approach (i.e. input data was not manually labelled before being trained on) to learning. PredNet [86], developed by Bill Lotter, Gabriel Kreiman and David Cox in 2016, employs a convolutional neural network that given a sequence of frames, can produce a predicted future frame (using both real and synthetic image sequences). Critically, PredNet is able to consider background and foreground scenery when predicting future frames, producing realistic results with some reduction in quality. It may therefore be possible to feed interaction and scene context data into a model similar to PredNet and generate future frames for an IRR system, masking incorrect predictions. Unfortunately, no work in this domain exists yet and is therefore a promising research direction.

8.2.4. Panoramas

Panorama images, rendered by the server application, may be an appropriate way to mask incorrect predictions, as well as make the client feel more responsive and increase the interaction space available to the user. 360-degree panoramic spheres can be rendered by the server application and delivered to the client. Rotational movements can easily be managed by the client and when prediction fails, the illusion of forward and backward movement could be generated by zooming (in and out of) the image on the client up to a certain threshold [63]. Like the image warping approach, this would fail for foreground and background objects moving within the scene and not linked to any interaction.

8.2.5. Multi-track prediction

The prediction module described earlier is only able to identify an incorrect prediction once it has received an update from the client application, and the server application relies on the prediction module for instructions and is unconcerned about what happens on the client (from an

interaction perspective). It is critical that the prediction module is informed as soon as possible of an incorrect prediction, since further mispredictions are likely to follow and will therefore result in the server application processing and rendering invalid predictions, ultimately increasing the experienced IL.

The client application (§7.2.1) is able to identify when an incorrect prediction message has been received by comparing the action performed with the predicted one, but must communicate errors to the prediction module as well as keep track of received messages and be careful to not discard those required further in the future.

The multi-track prediction scheme attempts to perform prediction for multiple paths and therefore minimise the potential for a misprediction. For instance, rather than predicting one step ahead, the system should predict one step ahead but for multiple possible directions. Consider Figure 8.1, where four interactions, performed sequentially, arrive on the prediction module from the client application. The figure shows each of these interactions with a current interaction position and a possible trajectory.

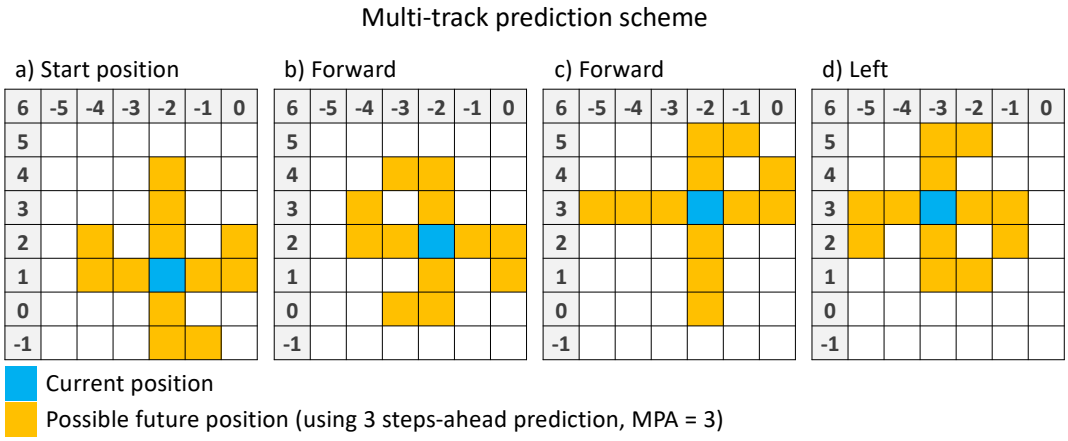


Figure 8.1 Multi-track prediction scheme. 4 possible user actions with MPA = 4.

In Figure 8.1a, the player/camera is at position {1, -2}, and can move either forward, back, left or right and since MPA = 3, the prediction module must therefore produce twelve predictions (represented by orange). This tells us that for each interaction performed on the client, the prediction module must produce $MPA * n$ predictions, where n is the number of possible interactions the user can perform. A prediction for the current (blue) position is not required, since it should have been created and predicted with the previous interaction to arrive on the prediction module. In b), c) and d) of Figure 8.1, the user has moved the camera forward, forward and left, to position {2, -}, {3, -2}, and {3, -3}, respectively. For each interaction that arrives on the prediction module, all $MPA * n$ predictions must be made; this is because previous predictions may, at this point, be invalid. For example, if the actions, “forward”, “backward”, “forward”, were performed, then the scene generated by the first forward action might be different than that generated on the second: an scene update may have occurred without input from the user – such as an object moving in a simulation.

On the other hand, if scene updates do not occur without direct input from the user, then previously predicted interactions remain valid and their resulting frames may be reused. In this case, it is not necessary to produce $MPA * n$ predictions per interaction, but rather:

$$numberofpredictionstogenerate = \frac{MPA * n}{2} + 1 \tag{8.1}$$

For example:

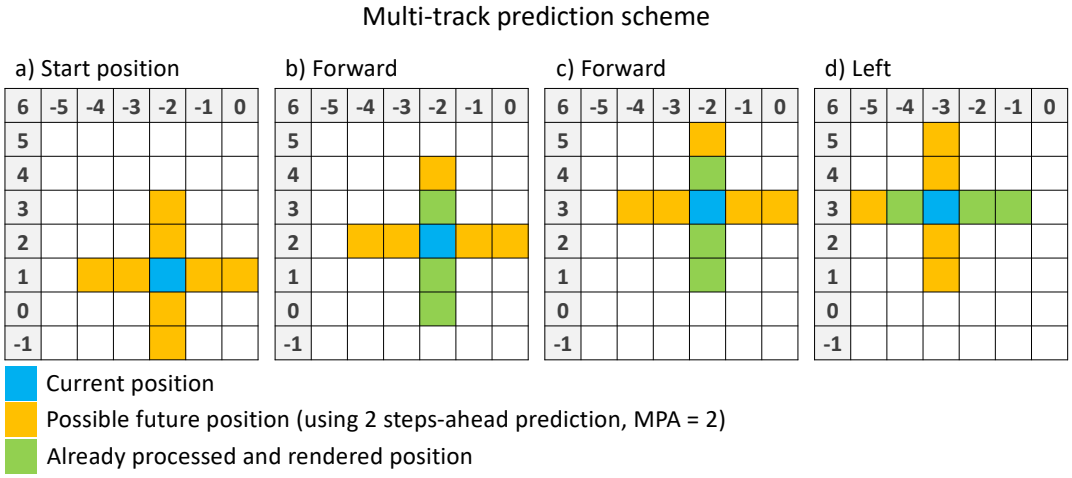


Figure 8.2 Multi-track prediction scheme with 2-steps ahead prediction. 4 possible user actions.

Figure 8.2 demonstrates how in a system where scene updates are entirely driven by user interaction, previous results can be reused, thereby lessening the required number of predictions therefore reducing the burden on the server application. Figure 8.2a shows that an action has just arrived on the prediction module from the client application, placing the user at the current (blue) position {1, -2}. From this position, the user might take one of four potential paths, all of which are predicted and sent to the server application. Next in b), another interaction has arrived, placing the user at position {2, -2}. In this case, the position has been previously been correctly predicted in a), and therefore the N-Gram model updates and new potential paths are generated. Since in this example scene updates are governed entirely by user interaction, previous results can be used. Positions {3, -2}, {2, -2}, {1, -2}, and {0, -2} were previously (green) predicted and therefore these predictions do not need to be sent to the prediction module, while positions {-4, 2}, {-3, 2}, {-2, 4}, {-1, 3} and {0, 3} have yet to be sent to the server and therefore, do need to be sent. As we can see with c) and d) of Figure 8.2, the number of predictions which need to be made for each interaction remains constant at 5 (orange). These predictions eventually arrive on the server application, which then performs rendering and finally forwards the results to the client.

8.3. The effect of a multi-track prediction strategy on IL due to misprediction

The fundamental idea is that by having the prediction module cover as many possible future interactions as possible, a correct future-result is likely to have been delivered to the client and

therefore hopefully lessens the impact of an incorrect prediction on IL. However, this approach raises a number of questions, such as:

1. Does the multi-track approach help to mitigate the effects of misprediction?
2. How does MPA affect the ability of the prediction module to mask IL?
3. Can the client application manage the large number of messages received from the server application?

To address these questions, the same client-predictor-server architecture as described in §7.2 was use, however the prediction module described in §5.3.3 was modified to predict a path for each of the possible interactions the user can perform, and for each path, predict n (MPA) steps ahead. This modification entailed the following:

> If an interaction has arrived on the prediction module and if a prediction for this interaction created, this original interaction is forwarded to the server application. A new set of predictions is then generated for this latest interaction:
>
> - For each step ahead to predict
>   - For each possible action the user can perform
>   * Create an empty prediction
>   * Assign the interaction to the prediction
>   * Assign the interaction number as being equal to the current interaction number + the step ahead being predicted + 1
>   * Generate a position for the new location and assign it
>   * Create and assign a key as described in §5.3
>   * Add prediction to queue

After generating a queue of predictions, the queue is filtered such that only predictions which have not already been sent to the server application remain. If, on the next iteration of the prediction loop, no interaction has been received, the prediction module sends each of the predictions, in turn, to the server application for processing, rendering and forwarding to the client application. After each prediction is transmitted, its key is added to a dictionary of previously sent predictions, which is used in the described filtering of predictions process.

All experimentation was conducted using the Unity3D PIRR system and as an initial test, the system was run using Multi-Track prediction with MPA = 1, N-Gram Order = 1 and NL set to 0. After the first experiment run, it became clear that the approach had a serious issue: IL was found to increase from the first interaction and continued to do so until all interactions had been simulated. The reason for this was found to be in relation to the server application: since a path for each possible action (of which there are four in this instance) is being predicted ahead of time, the prediction module generates many more messages which eventually arrive on the

server application for processing and rendering. As a result, the server application blocks all future messages until rendering is finished for the current one. Therefore, the system architecture was modified slightly such that multiple server applications could be operating simultaneously. In addition, the prediction module was modified to distribute the prediction messages across however many server applications are being used. The client application was adapted to accommodate results arriving from the various server applications, too. Figure 8.3 provides a high-level illustration of the new architecture.

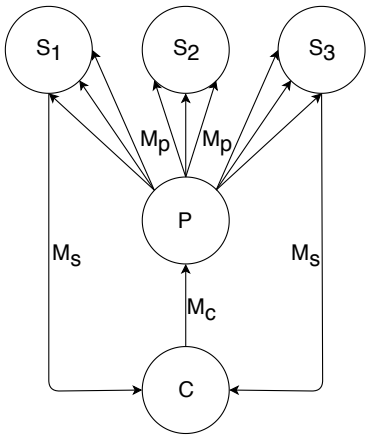


Figure 8.3 Modified Interactive Remote Rendering Unity3D system to use multiple server applications.

Experiments were then conducted using the newly modified Unity3D system with the with N-Gram Order set to one, various MPA values from 1 to 5 and NL values of 0ms, 50ms, 100ms, 200ms, 300ms and 400ms. All experiments were performed on a local machine (MacBook Pro running Windows in Bootcamp mode) with CSP prediction mode. The first experiment was performed with MPA = 1, NL = 0ms and with two server application instances, resulting in 0.08ms measured IL. This was found to be promising, since in previous experiments (for both simulator and Unity3D platforms) using the same parameters, IL has always been above 0, albeit below 2ms (see Figure 6.6 and Figure 7.6). The fact that such a near-zero value of IL was recorded is an indication that mispredictions were being covered and therefore their impact was being reduced. Further experiments were conducted using the remaining MPA and NL parameter values and each experiment was repeated five times. All these experiments were performed first with two server applications, and then repeated each time with a different number of server applications (2 to 10). With the number of server applications set to two, the Unity3D system is only able to use MPA = 1. If MPA is set any higher, IL begins to climb rapidly; the system is clearly unusable (see Figure 8.4), even with 0ms simulated NL, the Unity3D system fails at with just two MPA. Note that the apparent spike in the figure is an anomaly. For a period of about two weeks, AWS was experiencing issues where CPU load would spike at brief intervals, impacting the performance of the PIRR system and any other applications running on the server. While the spike was not present when running experiments on another machine, AWS was used regardless due to it ability to host many more server application instances and therefore allow for more experimentation.

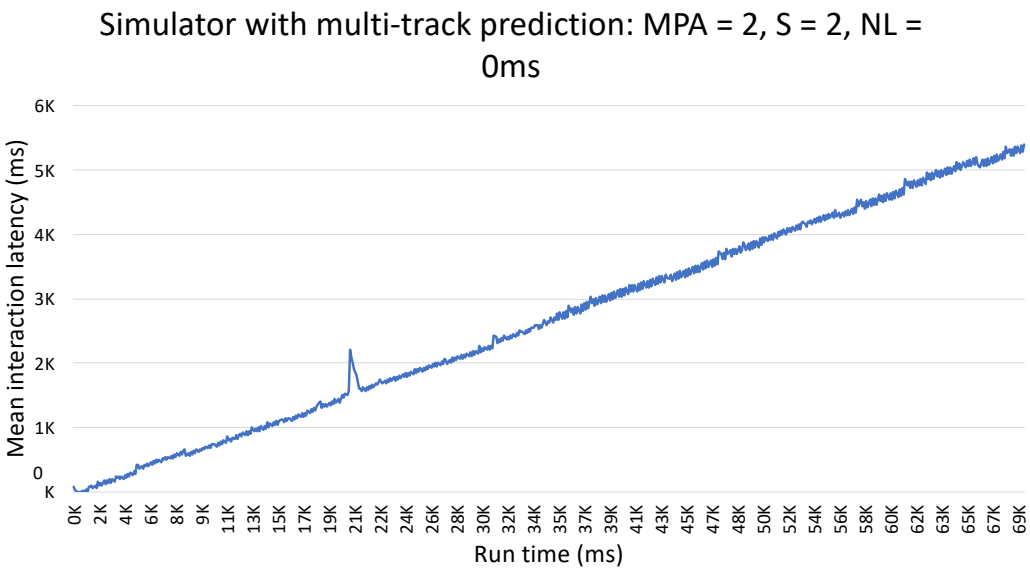


Figure 8.4 Interactive Remote Rendering Unity3D system with multi-track prediction failure.

The following figure illustrates a configuration of two server applications with MPA = 1 at various NL values and compares these results to their single-track Unity3D system counterpart measurements:

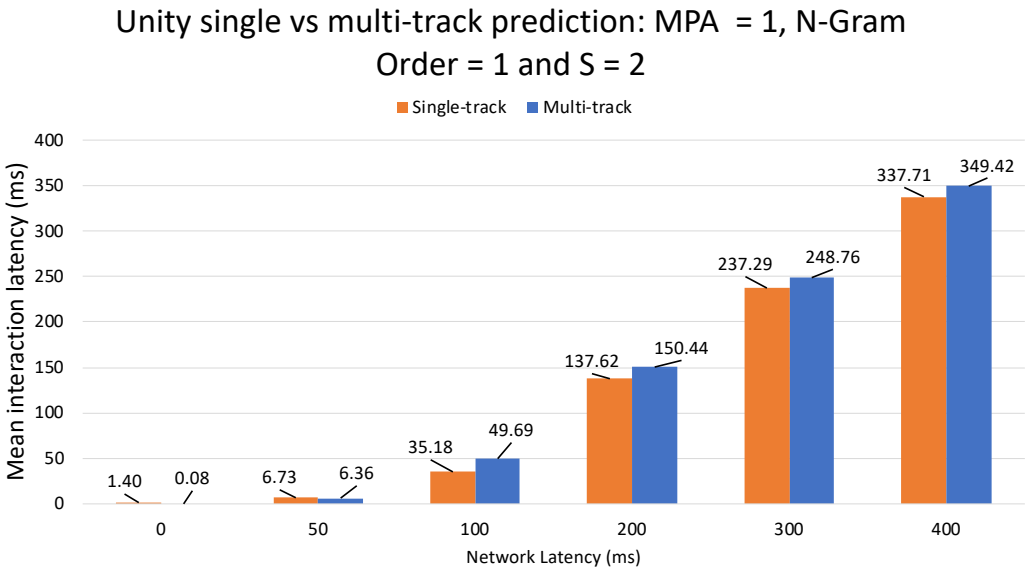


Figure 8.5 Interactive Remote Rendering Unity3D system comparison of single-track and multi-track prediction scheme results

As Figure 8.5 shows, single track out-performs the multi-track prediction scheme by about 10 to 15ms. When a greater number of server applications were employed, no change in mean IL was found. Increasing MPA for higher NL values did produce acceptable mean IL results, but the system was only able to operate up to 200ms of simulated latency. Increasing simulated NL would require a higher MPA value to mask IL, and this in turn requires a greater number of server applications to operate. Unfortunately, the system is not capable of operating the multi-track prediction scheme (see Table 8.1) with a MPA value greater than 3: too many messages pass through the system and the client application is unable to cope with the large

influx of these messages. The result is a rapid increase in IL as the client application struggles to process an ever-growing backlog of messages. More specifically, if for example the system generates 10 (SD) interactions per second and if an MPA of 5 is used and just 4 distinct actions α, can be used, the prediction module will produce up to 200 messages a second.

$$Numberofpredictedmessagespersecond = MPA * \alpha * SD \tag{8.2}$$

The client application will receive those 200 messages a second but unfortunately, in this case, it is unable to process them fast enough due to various delays such as decompression of arrived frames, and therefore a backlog forms.

Table 8.1 Mean IL measurements recorded from the Unity3D system using multi-track prediction. # Server applications represents the total number of instances of a single server application per experiment run. In all cases, the system was operated with CSP and on a local machine.

NL (ms)	MPA	# Server applications	Mean IL (ms)
0	3	6	0.08
50	3	6	0.30
100	3	6	2.02
200	3	6	11.87
300	3	7	93.82
400	3	7	268.57

Table 8.1 shows the lowest mean IL results recorded during experimentation at the various NL values require at least MPA = 3 and between 6 and 7 server applications. Unfortunately, as mentioned, increasing MPA to 4 resulted in extreme IL. Additionally, it is clear that mean IL increases dramatically across the simulated NL values. The reasons for these large changes in mean IL are unknown but is likely due to the instability of the system and the challenges of precise measurement at such low latencies.

8.4. Summary

This chapter has explored various methods for addressing the issue of misprediction in PIRR systems. Local rendering, image warping, prediction and panoramas were all briefly introduced as potential solutions and their issues discussed. However, these approaches do not adequately address the issues of misprediction, either due to the introduction of artefacts in images, the introduction of additional complications or the failure to take full advantage of the benefits of IRR. PredNet is the most promising of these approaches, but the early stage of the research makes it unclear how suitable it is. In terms of performance, local rendering and image warping would require significant resources on the client device and performance would therefore be dependant on the availability of those resources, while the panorama approach would likely also increase IL and use a significant amount of bandwidth. This is because in order to render the panorama, multiple virtual cameras would need to perform rendering from different angles. The resulting images from those virtual cameras would then need to be "stitched" together and then sent over the network to the client.

Multi-track prediction was then suggested as a way forward, and a framework and its implementation into PIRR systems was described. The system was experimented with at various NL and MPA values, but it was found that the system is unable to yield acceptable IL with a single server application instance. This was because of a backlog forming on the server application due to the single-threaded nature of Unity3D. As a result, the system was modified to utilize multiple server applications, but unfortunately suffers from stability issues, cannot deal with an MPA greater than 3 and requires between 6 and 7 server applications per experiment in order to achieve low mean IL measurements. With further experimentation and with server applications distributed over a number of EC2 instances, the multi-track approach would yield better results and is therefore worth further investigation.

Chapter 9. Client-Side Prediction vs Server-Side Prediction

9.1. Overview

The designing of the prediction module (described in §5.3.3) led to an interesting question: on which side of the network should the prediction module reside? In literature, this question does not appear to have been raised and therefore the focus of this chapter is to explore the benefits (if any, and with respect to IL) of running the prediction module on either the client-side (Client-Side Prediction (CSP)) or the server-side (Server-Side Prediction (SSP)) of the network. By having the prediction module and the client application execute on different machines, the client application will be able to operate without sharing its resources. Performing prediction on the server-side of the network will enable the prediction module to take advantage of the resource scalability of the cloud and not be impeded by the background application or processes running on the client machine. This will in turn result in greater performance of the system and therefore lower IL. However, CSP may provide lower IL since the prediction module will have access to interactions from the client sooner than it would have in an SSP configuration: the prediction module would need to wait $\frac{1}{2}$ NL before an interaction from the client arrives when using SSP, compared to 0ms when using CSP. For these reasons, both CSP and SSP configurations are explored.

9.1.1. Client-side prediction and server-side prediction

In modern networked video games, a common architecture is the "authoritative server" model. The primary aim of the authoritative server models is to mitigate cheating, and to reduce the delay between performing an interaction and seeing on-screen update, and therefore make user interaction feel as near instantaneous as possible [87]. In this architecture model, the server manages the true state of all entities within the game/visualization. User interactions performed on the client application are processed and rendered locally but the interactions are also sent to the remote server for "verification". If the server detects a discrepancy or an invalid action, an update is issued and the client "rolls back" to a previous state – a valid one dictated by the server; this can sometimes introduce "rubber banding" [88] which is when the client application is reset to an earlier point in time, leading to irritating and sometimes confusing results. In the game development world, this technique is known as CSP, since the client application essentially runs ahead of the server application. In this context, however, the term CSP is misleading since the client does not actually predict user interactions ahead of time – it merely allows them to execute locally and has the server validate them.

In this research, it is therefore important to make a clear distinction as to what is meant by CSP, as well as to define SSP:

CSP: The prediction module is located on the same side of the network as the client application and therefore shares resources available one the client device.

SSP: The prediction module is located on the same side of the network as the server application and therefore shares resources available on the server. There may be clear benefits to hosting the prediction module on the server. For example:

- More resources (computation, storage, memory) are likely to be available to the prediction module on the server, than on a local thin client device.
- A decrease in (or unstable) local resources on the client device will not impact the prediction module.
- More powerful prediction modules may be employed (or combined) on the server, than on the client device.
- There could be a feedback link implemented between the rendering application and the prediction module; this may allow the prediction model to consider geometry, textures, regions of interest, etc., and therefore raises the possibility of using prediction modules not possible when operating on the client.

However, there may be benefits to hosting the prediction module on the client, too:

- Incorrect predictions can be realised sooner (on the prediction module), since updates from the client do not have to travel across a network.
- External hardware, such as eye or gaze tracking might be useful for improving prediction but produces a significant amount of measurements and therefore may require unreasonable amounts of bandwidth.
- By having the prediction module on the client, hardware data can be fed directly into the prediction model and therefore may not be required to be transmitted over the network, resulting in lower bandwidth requirements.

The main focus of this thesis is IL and therefore in an effort to understand the effect of the two approaches on IL, the PIRR simulator architecture was modified slightly.
The modifications were made to the prediction module, server application and the initialization message sent on system start up. When CSP is used, a routine instructs that messages arriving at the prediction module must bypass the latency simulators, without introducing any delays. On the server application, arriving messages must be delayed and are therefore passed to the latency

Table 9.1 The above shows the modification made to the config in order to allow Client-Side Prediction or Server-Side Prediction mode.

```
[Experiment]
# Previously described properties
use_csp = true
```

simulator. On the other hand, when SSP is to be used, messages are instructed to pass through the latency simulator on arrival at the prediction module, introducing a delay, and on the server application, the latency simulator is bypassed entirely – just as the prediction module is in the CSP configuration. The initialization message was modified with a simple Boolean flag, indicating whether CSP or SSP is to be used. Figure 9.1 makes clearer the placement of the latency module in order to simulate either CSP or SSP.

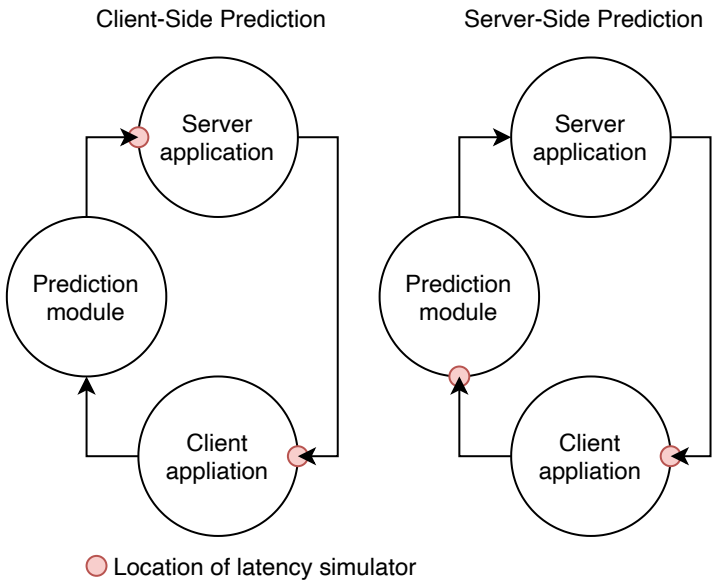


Figure 9.1 Client-Side Prediction (CSP) vs Server-Side Prediction (SSP).

In order to tell the system to use either CSP or SSP, the experiment section of the configuration file was modified with the addition of a “use_csp” property:

9.2. Experimental setup

In the following sections, CSP and SSP are compared. Comparisons are first made with respect to N-Gram Order, and then for MPA. In each case, four configurations are explored:

- Simulator CSP vs simulator SSP
- Unity3D CSP vs Unity3D SSP
- Simulator CSP vs Unity3D CSP
- Simulator SSP vs Unity3D SSP

In all experiments except for when operating over WAN, latencies from Table 6.1 were simulated and injected. When operating over WAN, the client application was run on a device located in Cambridge, UK, while the server application was hosted remotely on an Amazon EC2 g2.2xlarge instance located in Northern California, USA. In all experiments where prediction was used, the interaction template was fed into the client application, and interactions were performed automatically. Each NL/MPA/Order experiment configuration was repeated 5 times and in the case of WAN, experiments were repeated 5 times during morning, afternoon and evening; this was done to counter irregular network conditions as well as peak-hour traffic and broadband throttling.

All experiments using the simulator were performed using 30ms simulated render delay, while when using Unity3D, rendering was managed entirely by Unity and therefore is less controllable and more variable.

9.3. N-Grams

Earlier (§6.3.1), an increase in N-Gram Order (interaction history), when used to make predictions, was shown to be associated with higher IL: the greater the Order, the larger the mean IL, which is due to extended recovery times from incorrect predictions and the increased likelihood of an incorrect prediction due to too much history being used.
However in all those experiments, the CSP was used and it remains to be seen if the same results can be expected when running the PIRR system using a SSP configuration and therefore, the various experiments were performd and the system was evaluated.

The following table (Table 9.2) describes four groups of experiment configurations. Each mode/NL/N-Gram Order configuration was repeated 5 times, bringing the total number of experiments performed to 700. After completing all experiment runs, the results were grouped by repeat experiment, then by NL, followed by N-Gram Order and then averaged.

Table 9.2 Experiment parameters used for comparing CSP and SSP. Each configuration is composed of 150 experiment runs: 6 different latencies, 5 N-Gram Orders, repeated 5 times each.

Group	Platform	Mode	Network Latency (ms)	N-Gram Order	Repeats	Total
A	Simulator	CSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5	5	175
B	Simulator	SSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5	5	175
C	Unity3D	CSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5	5	175
D	Unity3D	SSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5	5	175

In all experiments, MPA was set to 1 and when simulated, rendering was set to 30ms.

9.3.1. Simulator CSP vs SSP

The aim of the following experiments was to determine if there is any difference between CSP and SSP modes using the simulator, with respect the N-Gram Order. After performing the experiments described in groups *A* and *B* of Table 9.2, results were collected and analysed. For each NL/Order group, all repeat experiments were grouped and averaged to produce a single set of mean IL measurements. In each latency group (Figure 9.2), the results of the experiments show that IL increases with N-Gram Order, regardless of which side of the network the prediction module is hosted on. Across all latencies, when the simulator is run with a CSP mode, IL is shown to be fractionally less than when run with SSP: the largest difference was found to be just over 1 millisecond.

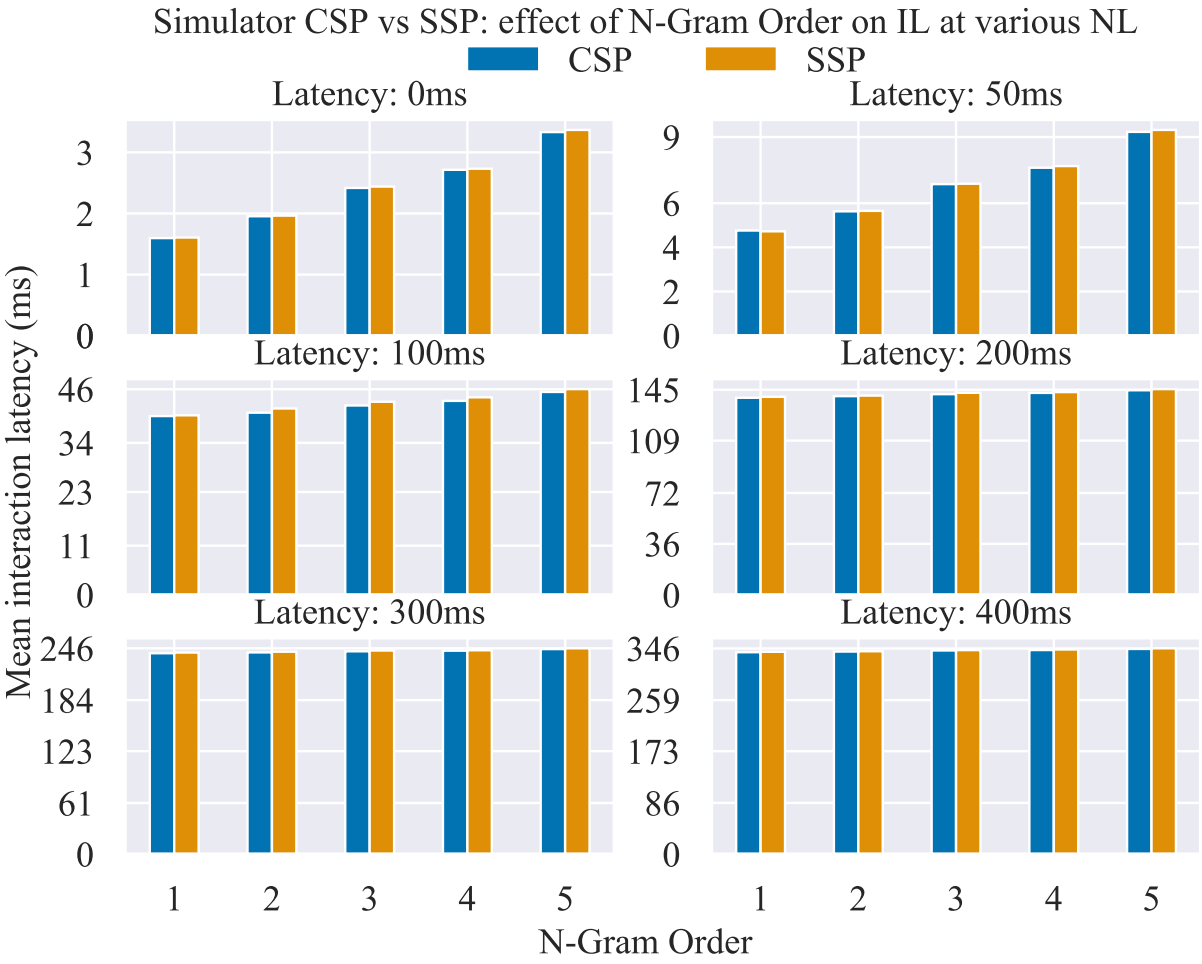


Figure 9.2 Effect of N-Gram Order on Interaction Latency using the Simulator: Client-Side Prediction vs Server-Side Prediction

On inspection, the results of both CSP and SSP experiments look very similar. However, it would be interesting to determine whether or not any difference between the two configurations is constant, or changes according to N-Gram Order as well as NL. In order to understand this, the results described in Figure 9.2 were compared at each N-Gram Order and NL. For instance, the mean IL measured at N-Gram Order = 1 and NL = 0ms for CSP is subtracted with the same measurement for SSP. This difference is represented by $\Delta\overline{IL}$.

Figure 9.3, describes the $\Delta\overline{IL}$ values (in milliseconds) for N-Gram Orders 1 to 5. In all cases, CSP measurements were found to be lower than their SSP counterparts. However, from the plot, it can be seen that the maximum difference between CSP and SSP measurements was just over 1ms, a negligible value. This observation was found in all NL groups.

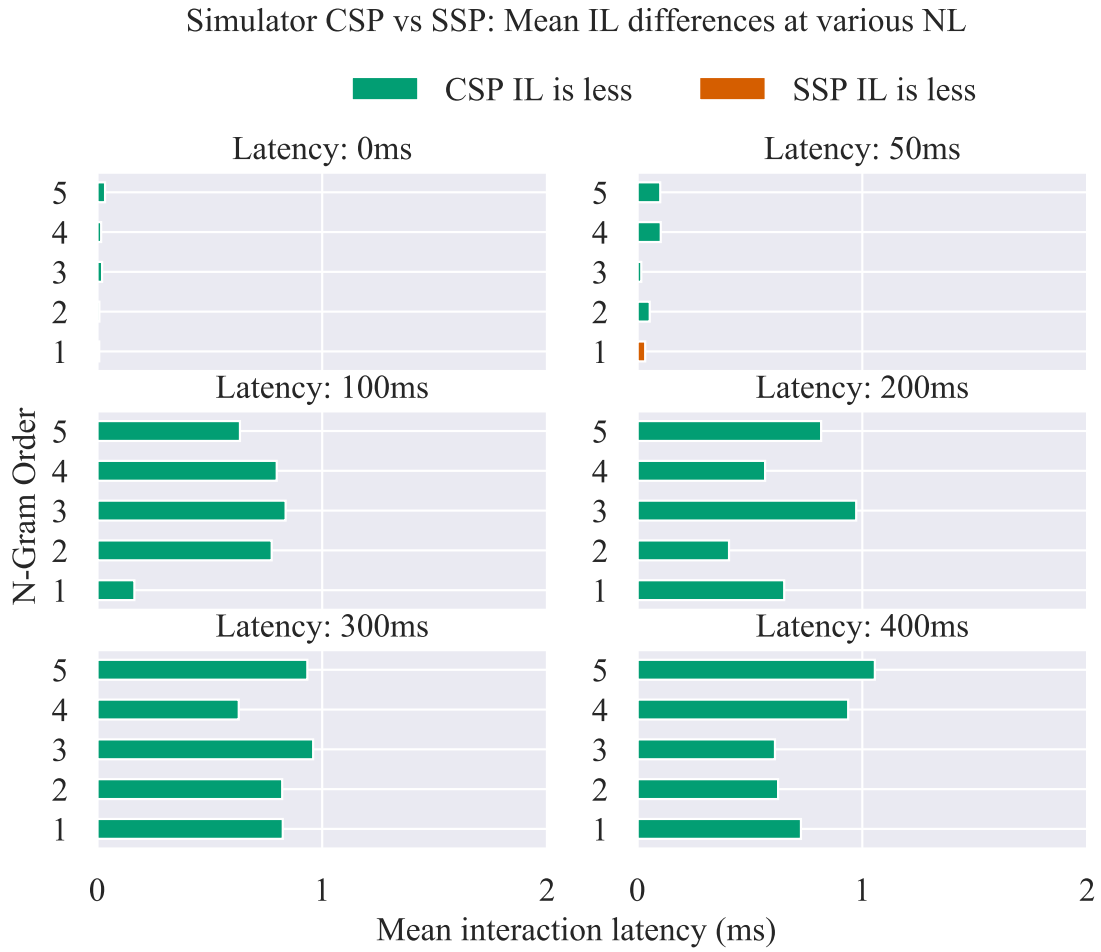


Figure 9.3 Simulator CSP vs SSP. Mean differences at various N-Gram Orders and Network Latencies. MPA = 1.

The same set of experiments were then repeated, but instead of simulating latency, the simulator was run over a WAN. Similarly, each experiment used a different N-Gram Order, was repeated 5 times. This was repeated morning, afternoon and evening and after collecting all results, they o were averaged. As can be seen in Figure 9.4, CSP again performs better over WAN, this time by a more pronounced margin. While the $\Delta\overline{IL}$ between CSP and SSP for these experiments, presented in Table 9.3, vary across N-Gram Orders, it is clear that on average, IL can be more than 10ms higher when running a SSP configuration. The sporadic $\Delta\overline{IL}$ is likely due to the instability of the network connection between Cambridge and Northern California, and the challenges of performing such small measurements.

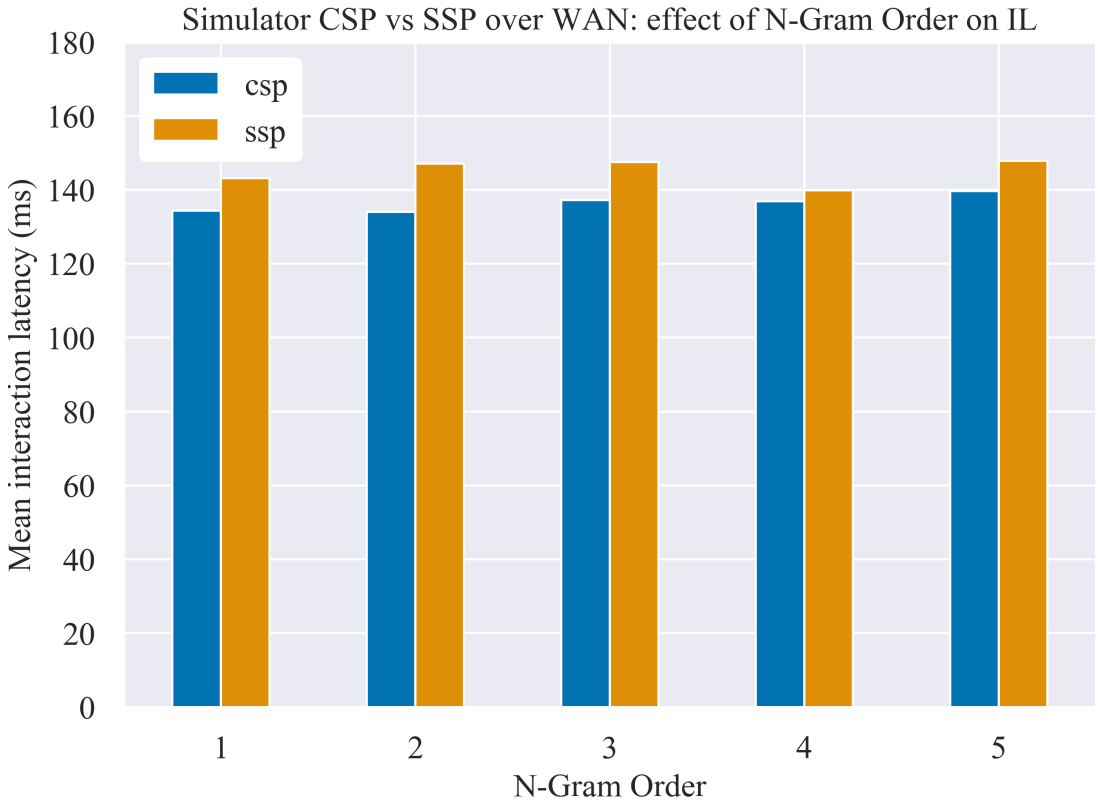


Figure 9.4 Simulator CSP vs SSP using WAN and the effect of N-Gram Order on IL.

Table 9.3 CSP vs SSP over WAN. The table shows the various N-Gram orders tested (1 to 5) and their effect on IL in either a CSP or SSP configuration. In all cases, CSP is lower than SSP. Numbers represent the mean IL difference ($\Delta\overline{IL}$) between CSP and SSP.

Order	1	2	3	4	5
$\Delta\overline{IL}$	8.196	12.212	9.123	3.148	10.894
Configuration with lowest IL	CSP	CSP	CSP	CSP	CSP

9.3.2. Unity3D CSP vs SSP

All of the experiments in §9.3.1 were repeated using the Unity3D PIRR system, rather than the simulator (see Table 9.2 Groups *C* and *D* for experiments and parameters). Results were processed identically to those in §9.3.1 and Figure 9.5 presents the results of the experiments.

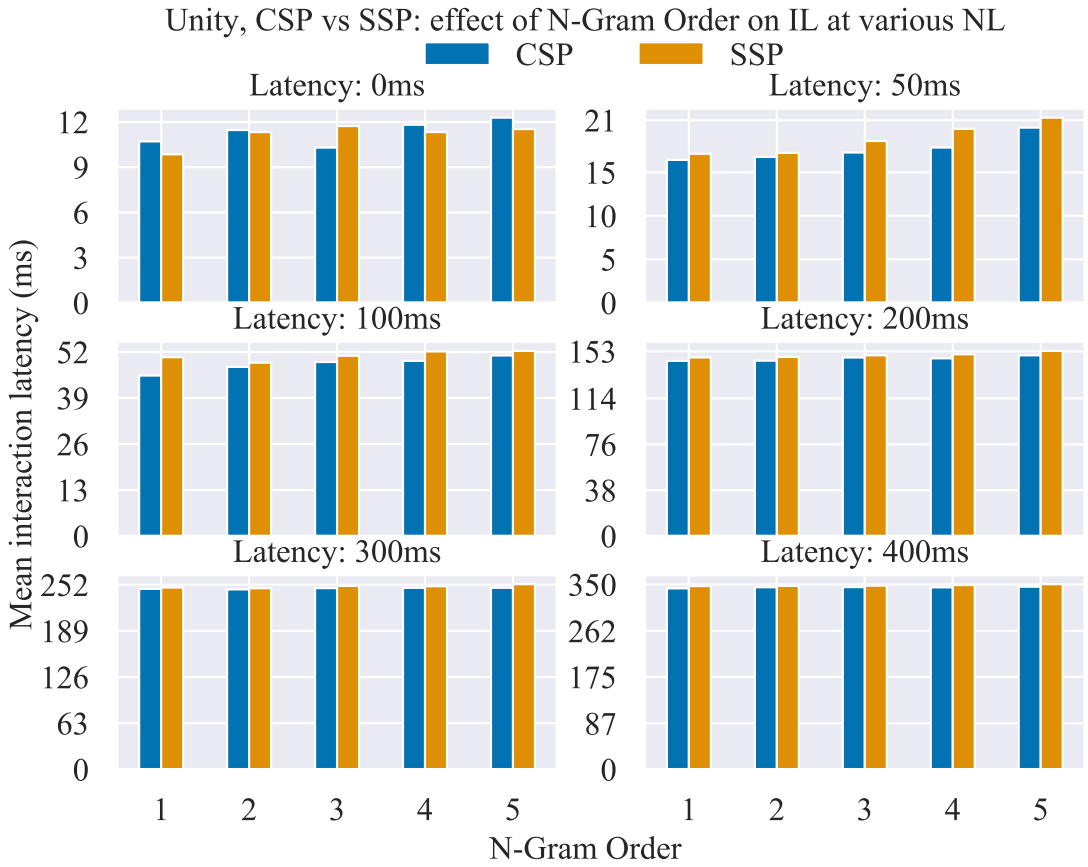


Figure 9.5 Unity3D CSP vs SSP using WAN and the effect of N-Gram Order on IL.

Similar to the results from the simulator (Figure 9.2), Figure 9.5 shows that when using the real-world PIRR system, CSP also offers lower IL than when using SSP. As can be seen, IL increases with both NL and N-Gram Order for each NL group. The mean differences between CSP and SSP experiment results were calculated for each N-Gram Order presented in Figure 9.5 and when inspecting the data, $\Delta\overline{IL}$ is shown to increase with NL and this is illustrated in Figure 9.6. For these experiments, the smallest $\Delta\overline{IL}$ occurs when NL = 0ms and $Order$ = 1 with a value of 0.852ms. On the other hand, the largest value of 4.742ms occurs at $Order$ = 5 and NL = 400ms, suggesting that $\Delta\overline{IL}$ increases with NL when using the Unity3D platform.

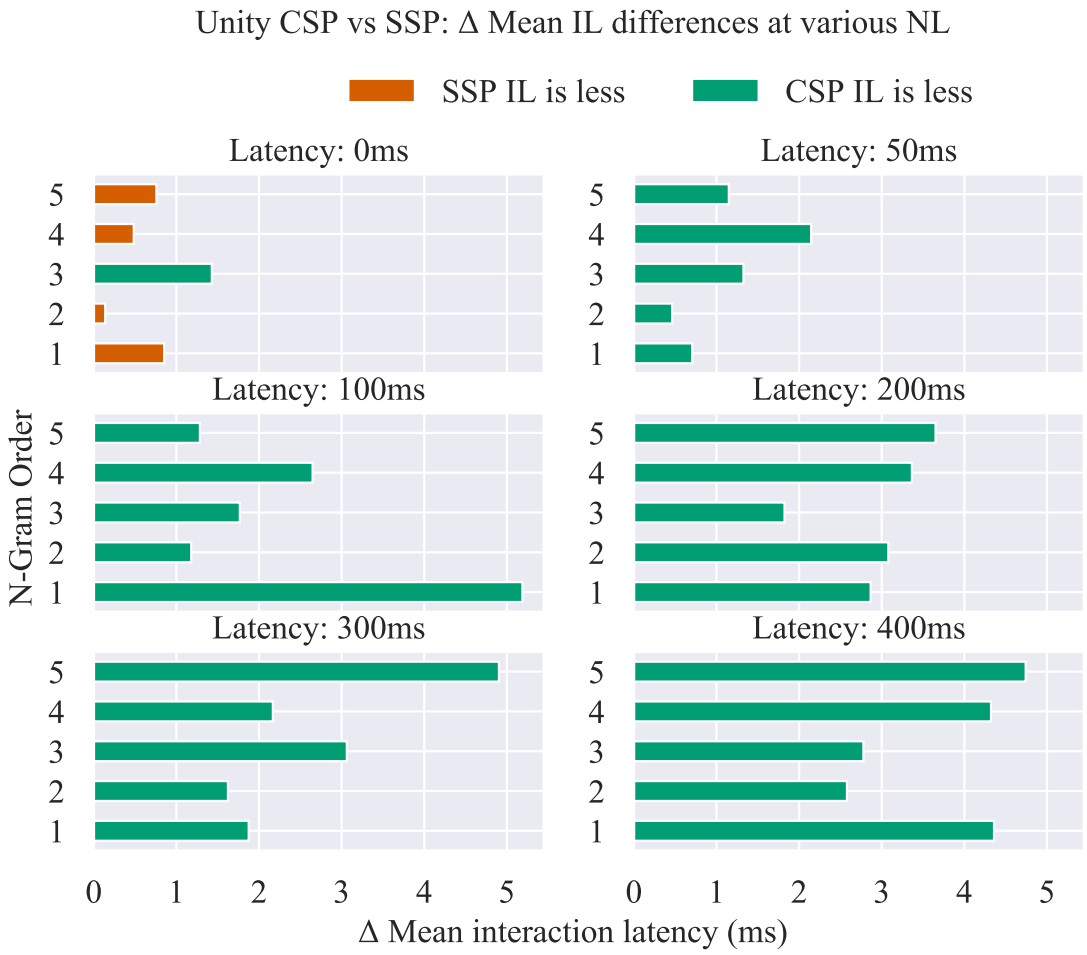


Figure 9.6 Unity3D CSP vs SSP Δ mean IL differences at various NL and N-Gram Orders.

The same N-Gram Orders, MPA and rendering delay were used in additional experiments run using a WAN. The results indicate that again, like the simulator, the Unity3D PIRR system also operates with lower IL when using the CSP mode, show in Figure 9.7.

In terms of $\Delta\overline{IL}$ between CSP and SSP modes using WAN, measurements surprisingly appear unaffected by N-Gram Order (Table 9.4). However, this may well be due to the variation in NL when using WAN, compounded by the fact that there is more variance in render delay when using Unity3D compared with the simulator.

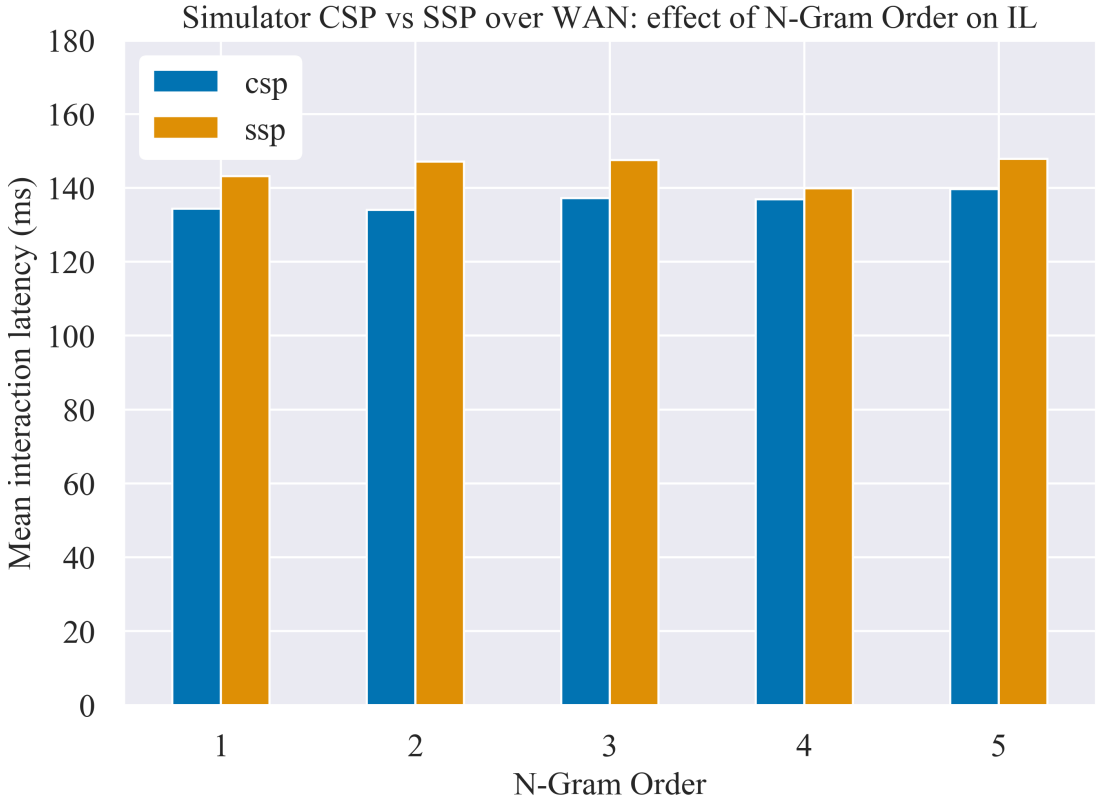


Figure 9.7 Unity3D system CSP vs SSP. Effect of N-Gram Order on IL over WAN.

Table 9.4 Unity CSP vs SSP over WAN. The table shows the various N-Gram orders tested (1 to 5) and their effect on IL in either a CSP or SSP configuration. In all cases, CSP is lower than SSP. Numbers represent the mean IL difference ($\Delta\overline{IL}$) between CSP and SSP.

Order	1	2	3	4	5
$\Delta\overline{IL}$	0.615	1.919	1.855	0.769	1.877
Configuration with lowest IL	CSP	CSP	CSP	CSP	CSP

9.3.3. Simulator CSP vs Unity3D CSP

The CSP experiment results gathered from experiments (Groups *A* and *C*, Table 9.2) conducted in §9.3.1 and §9.3.2 were reused so that the simulator and Unity3D could be compared. These comparisons (for each latency and N-Gram Order) are illustrated in Figure 9.8. In all experiments, the simulator yielded lower IL. IL also increased with both NL and N-Gram order. $\Delta\overline{IL}$ was then calculated for each N-Gram Order and NL. Figure 9.9 illustrates these results. Curiously, it was found that $\Delta\overline{IL}$ appears to be inversely proportional to NL. In other words, as NL increases, $\Delta\overline{IL}$ appears to decrease. However, this is likely due to a combination of fluctuations in rendering time (on Unity3D) and the difficulty of accurately simulating delays when using Thread.Sleep() (due to time-slicing on the CPU having a resolution of approximately 15m, as mentioned earlier). The highest $\Delta\overline{IL}$ was registered at 11.67ms for NL = 50ms and N-Gram Order = 1, while the lowest $\Delta\overline{IL}$ was 0.49ms, measured at NL = 400ms and N-Gram Order = 5.

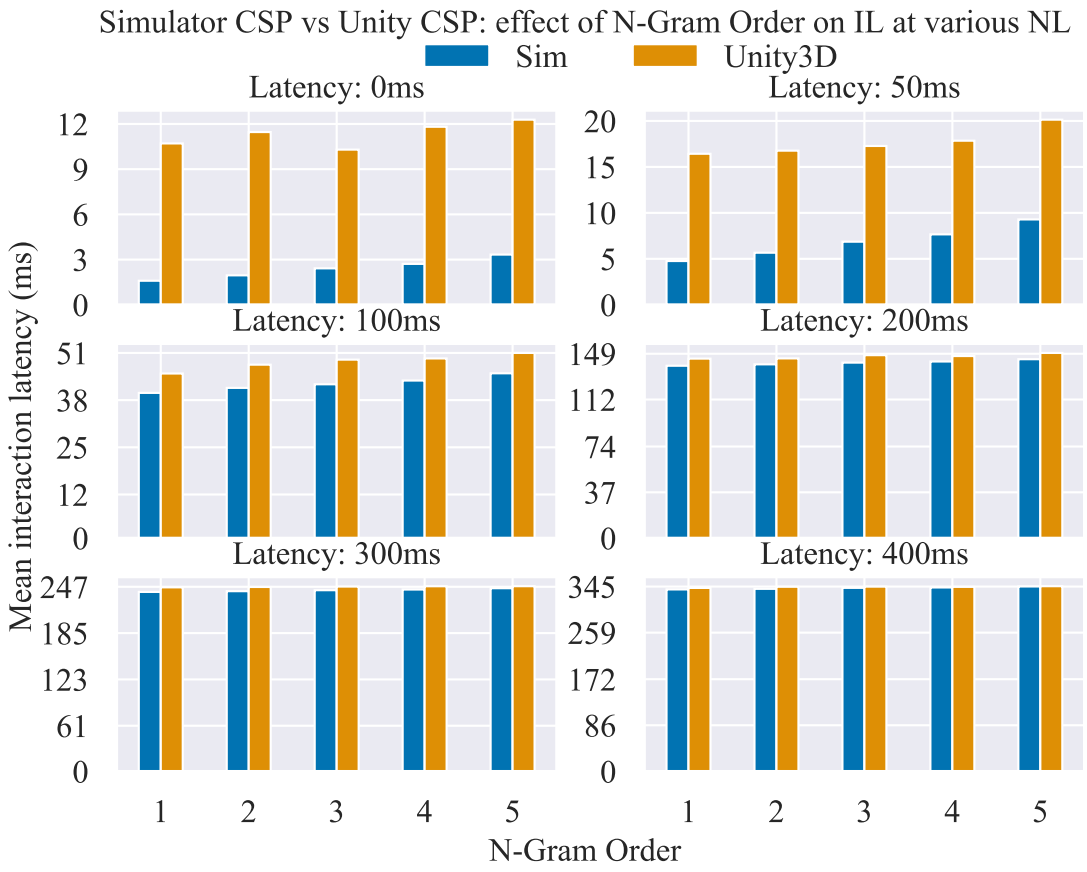


Figure 9.8 Simulator vs Unity3D CSP. The effect of N-Gram Order on IL at various NL.

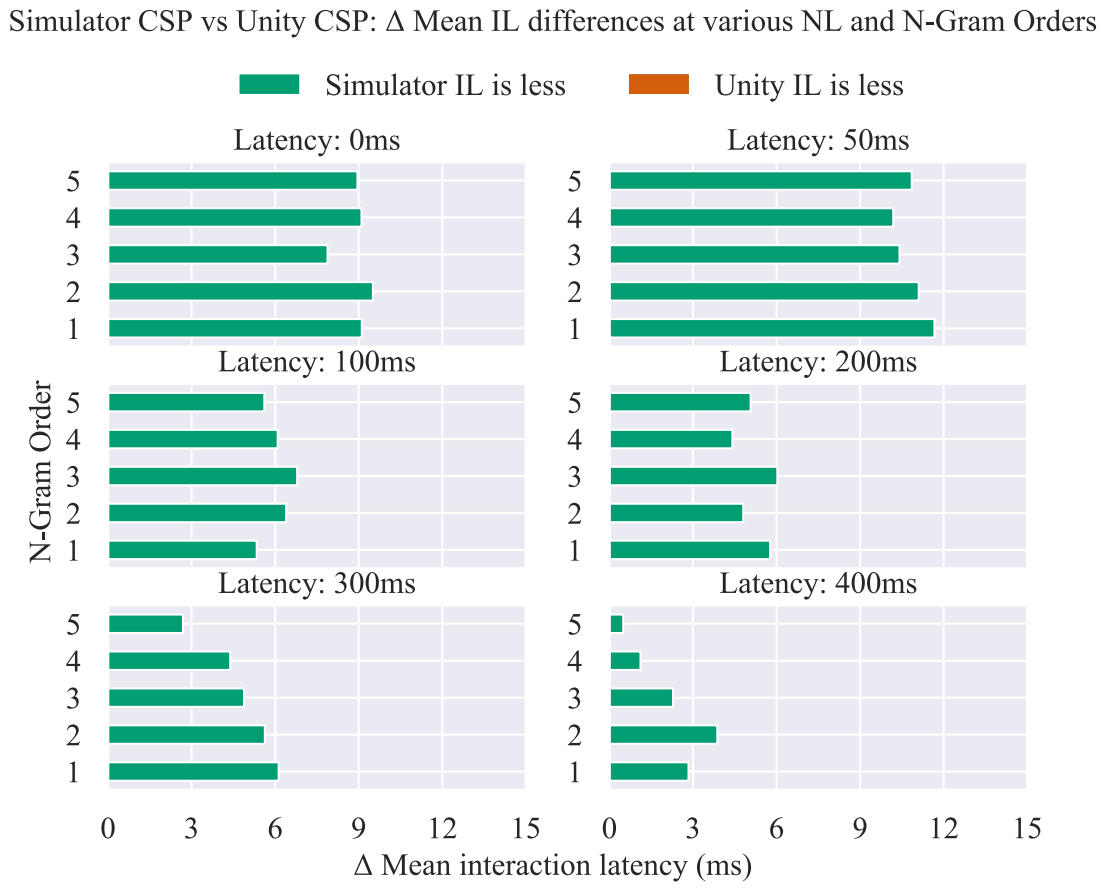


Figure 9.9 Simulator vs Unity3D CSP Δ mean differences. N-Gram Orders.

9.3.4. Simulator SSP vs Unity SSP

In this section, results obtained from SSP experiments when using the simulator and the Unity3D PIRR system are compared. The relevant configurations for these results can be found in Table 9.2 Groups *B* and *D*. After compiling the respective results and analysing the data, Figure 9.10 was generated. Just as with §9.3.3, the plot shows that the simulator produces lower mean IL values than the Unity3D PIRR system, when using SSP.

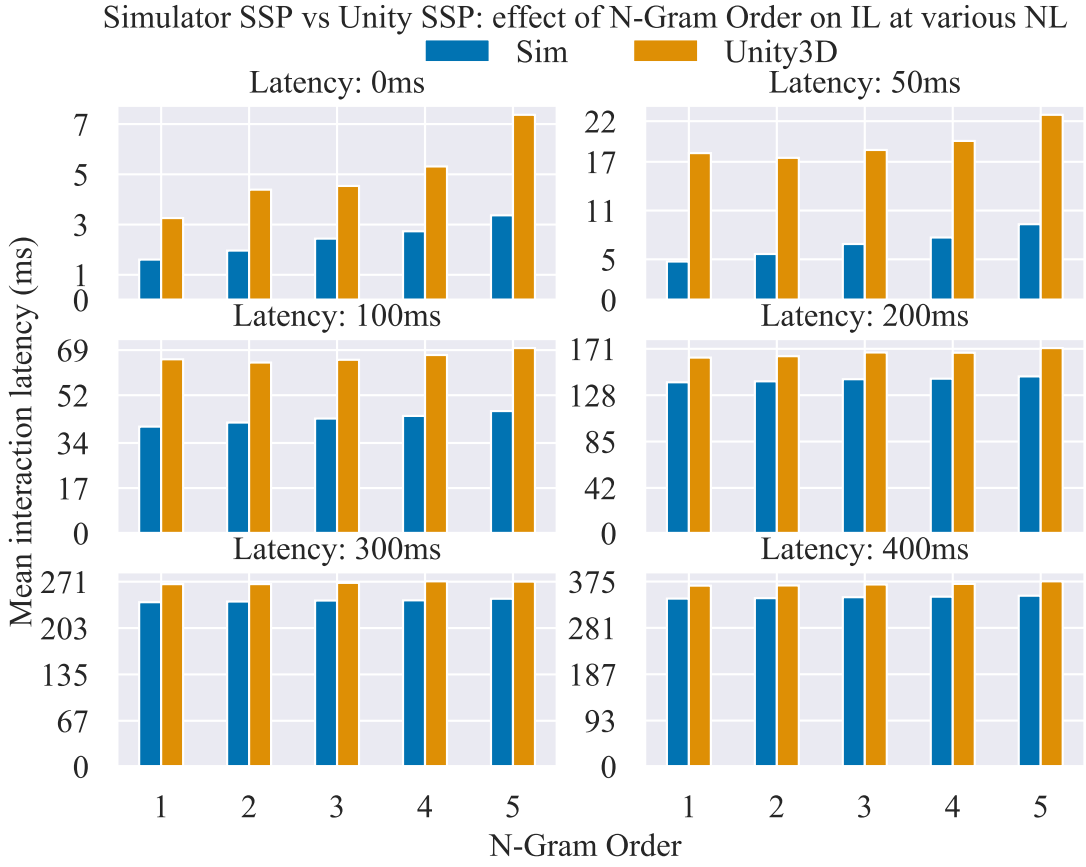


Figure 9.10 Simulator vs Unity3D SSP: Effect of N-Gram Order on IL at various NL.

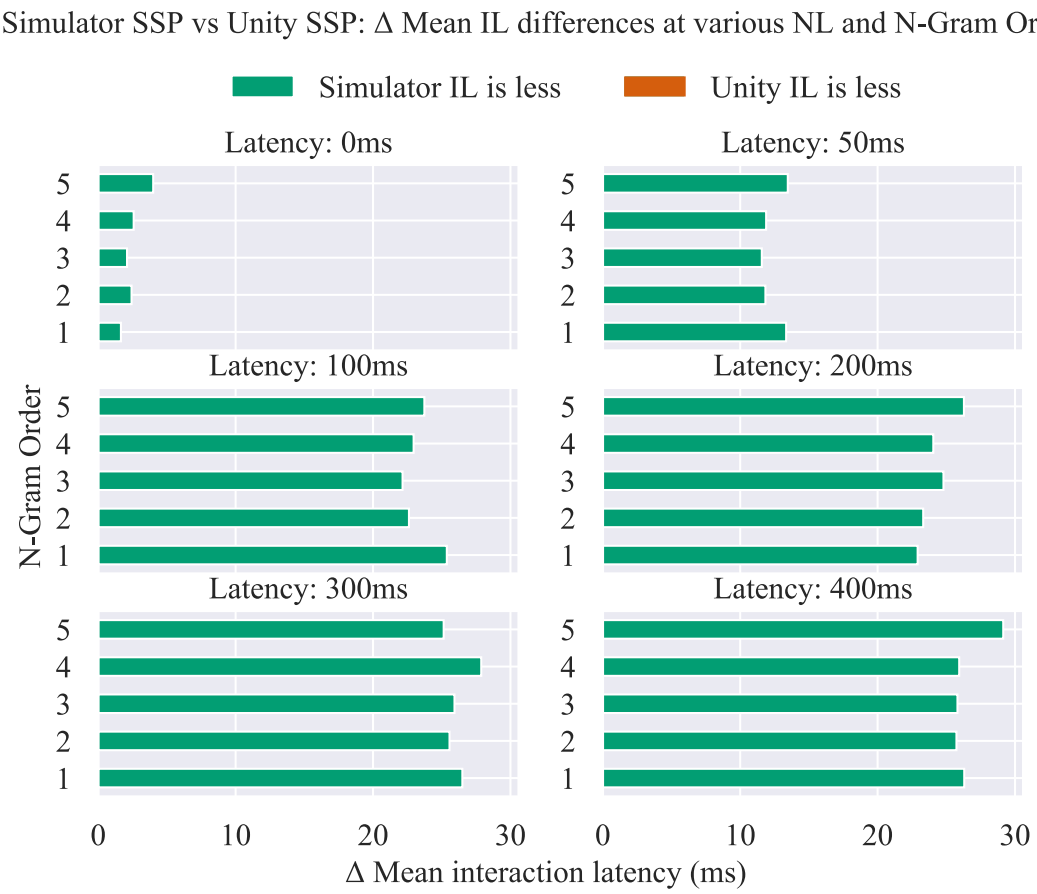


Figure 9.11 Simulator vs Unity3D SSP. Δ Mean differences. N-Gram Orders.

Figure 9.11 shows the $\Delta\overline{IL}$ values for the above experiment. From the figure, it can be seen that in all experiment, the simulator resulted in lower mean IL. The minimum $\Delta\overline{IL}$ was measured at NL = 0ms and N-Gram Order = 1, with a recorded value of 1.66ms, while the maximum measured $\Delta\overline{IL}$ was found to be 29.14ms at NL = 400ms and N-Gram Order = 5. This is interesting as is shows that the $\Delta\overline{IL}$ measured using SSP is higher than when using CSP, potentially further indicating that CSP results in slightly lower mean IL.

9.4. MPA

Using the simulator, MPA was earlier found to positively impact IL up to a certain point relative to a simulated NL. However, it is unknown whether CSP or SSP provides any benefits to IL with respect to MPA. In order to evaluate this in both the simulator and Unity3D PIRR system, the experiments in §9.3 were repeated with MPA parameters being updated, rather than N-Gram Order.

First, experiments were performed using the simulator. One set of experiments used a CSP and another set used the SSP configuration. The results from these experiments were then compared. The same experiments were then performed using the Unity3D PIRR system. Next, the results collected though experiments with the simulator and the Unity3D platforms are compared. In all cases, MPA's 1 to 10 were evaluated at the 6 different latencies from Table 6.1. In all

experiments, N-Gram Order was set to 1 and as indicated in §9.2, render delay was set to 30ms for simulator experiments.

Table 9.5 describes the same four groups of experiment configurations as presented in 9.2, but with N-Gram Order replaced by MPA. Each mode/NL/MPA configuration was repeated 5 times, bringing the total number of experiments performed to 1400. After completing the all experiment runs, the results were grouped according to repeat experiment, NL and MPA. Results were then averaged. In addition, difference between CSP and SSP results are calculated in order to determine whether or not the effect of either configuration grows or diminishes according to NL and/or MPA. This difference is represented by $\Delta\overline{IL}$.

Table 9.5 This table describes the various experiments conducted using the simulator and the Unity3D PIRR system. Various latencies and MPAs were evaluated in both CSP and SSP modes and each experiment was repeated 5 times.

Group	Platform	Mode	Network Latency (ms)	MPA	Repeats	Total
A	Simulator	CSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	5	350
B	Simulator	SSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	5	350
C	Unity3D	CSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	5	350
D	Unity3D	SSP	0, 50, 100, 200, 300, 400, WAN	1, 2, 3, 4, 5, 6, 7, 8, 9, 10	5	350

The following sub-sections explore configurations for:

- Simulator CSP vs Simulator SSP
- Unity3D CSP vs Unity3D SSP
- Simulator CSP vs Unity3D CSP
- Simulator SSP vs Unity3D SSP

9.4.1. Simulator CSP vs SSP

For the following experiments, the simulator was run with the parameters described in Table 9.5 groups *A* and *B*. In a Figure 9.12, it can be seen that CSP and SSP modes yield nearly identical results. In all subplots except for where NL = 0ms, SSP appears to offer slightly lower IL towards higher MPA. However, when inspecting the $\Delta\overline{IL}$ between CSP and SSP measurements for the various NL, it can be seen that a) $\Delta\overline{IL}$ increases with MPA, b) $\Delta\overline{IL}$ remain roughly the same across NL and c), SSP provides lower IL from about MPA = 5 upwards, where

measurements range from less than 1ms to approximately 11ms. Figure 9.13 presents the $\Delta\overline{IL}$ measurements of IL.

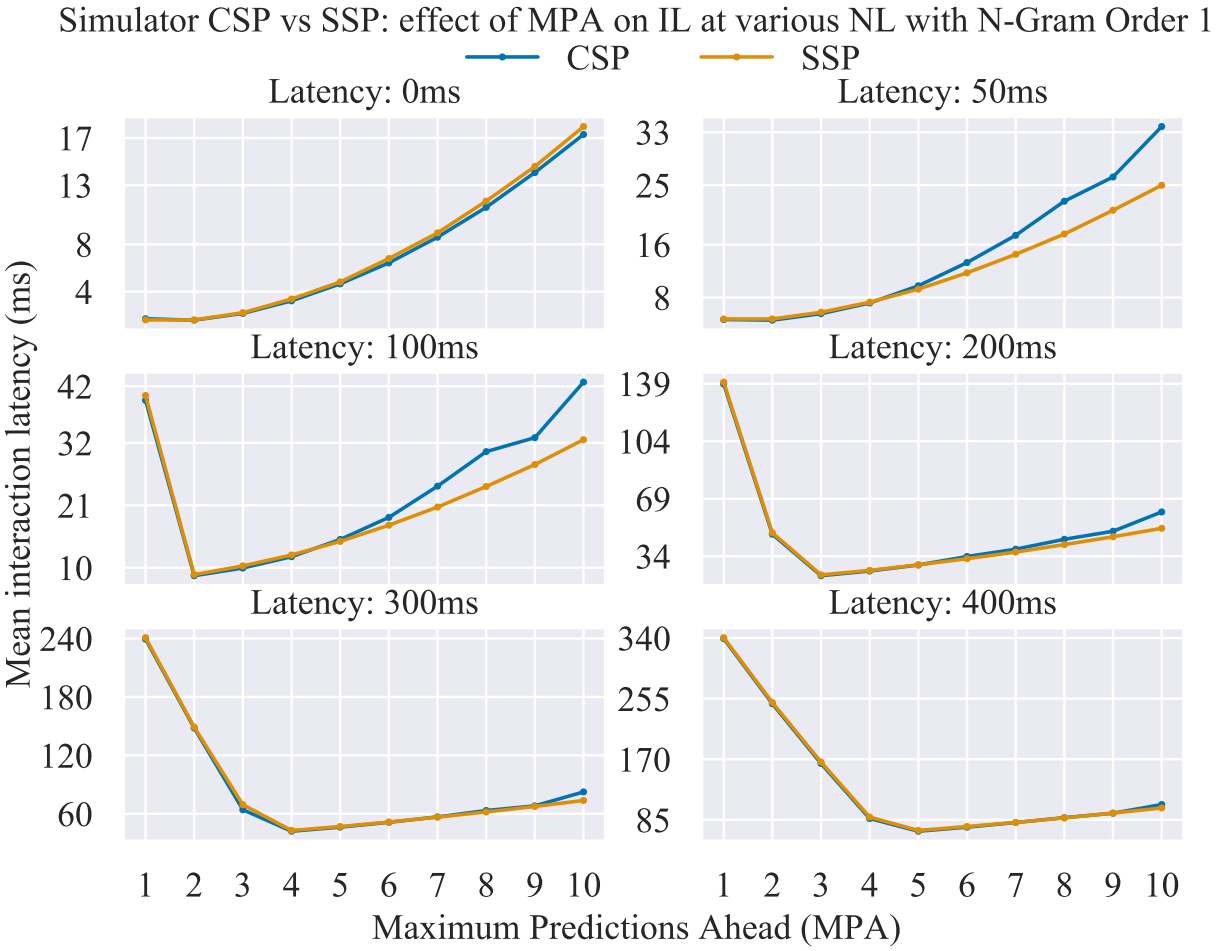


Figure 9.12 Simulator CSP vs SSP: The effect of MPA on IL at various NL.

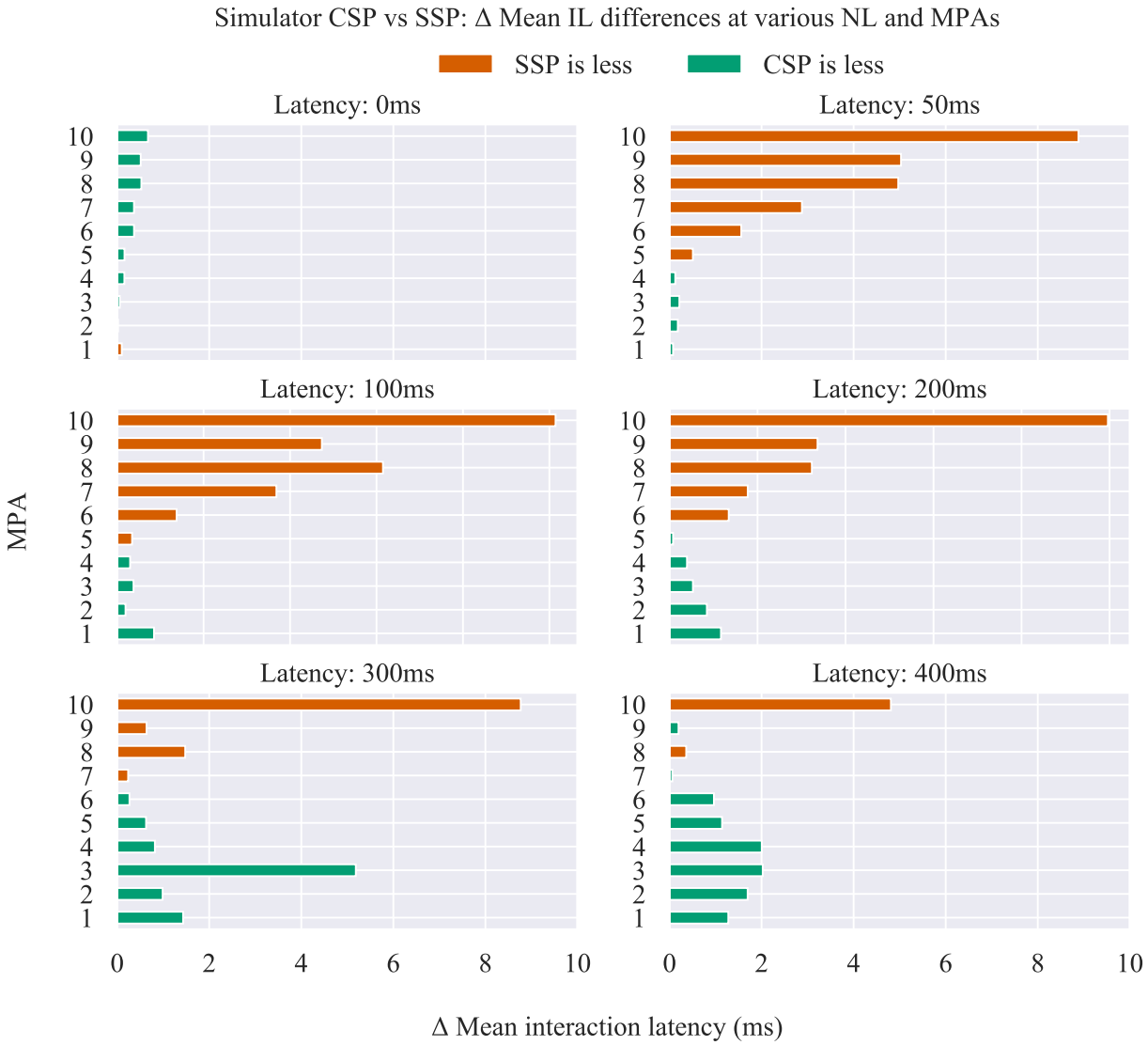


Figure 9.13 Simulator CSP vs SSP: Δ Mean IL differences at various NL and MPAs

SSP appears to demonstrate an advantage over CSP: with each increase of MPA, an increasingly lower IL is observed. This is likely because in SSP mode, where there is no network delay between the prediction module and server application, the server application is provided with predictions for processing without a delay. On the other hand, with the CSP mode (where there is a delay between the prediction module and the server application) the server application must wait for 1⁄2 NL before receiving results from the prediction module. This difference is important, since when operating in CSP mode, the client device struggles to cope with the large volume of messages arriving for processing and the burden of running the prediction module further strains local compute resources.

After running the simulator using simulated latencies, the same experiments described here were instead performed over WAN, between Cambridge, UK and Northern California, USA.

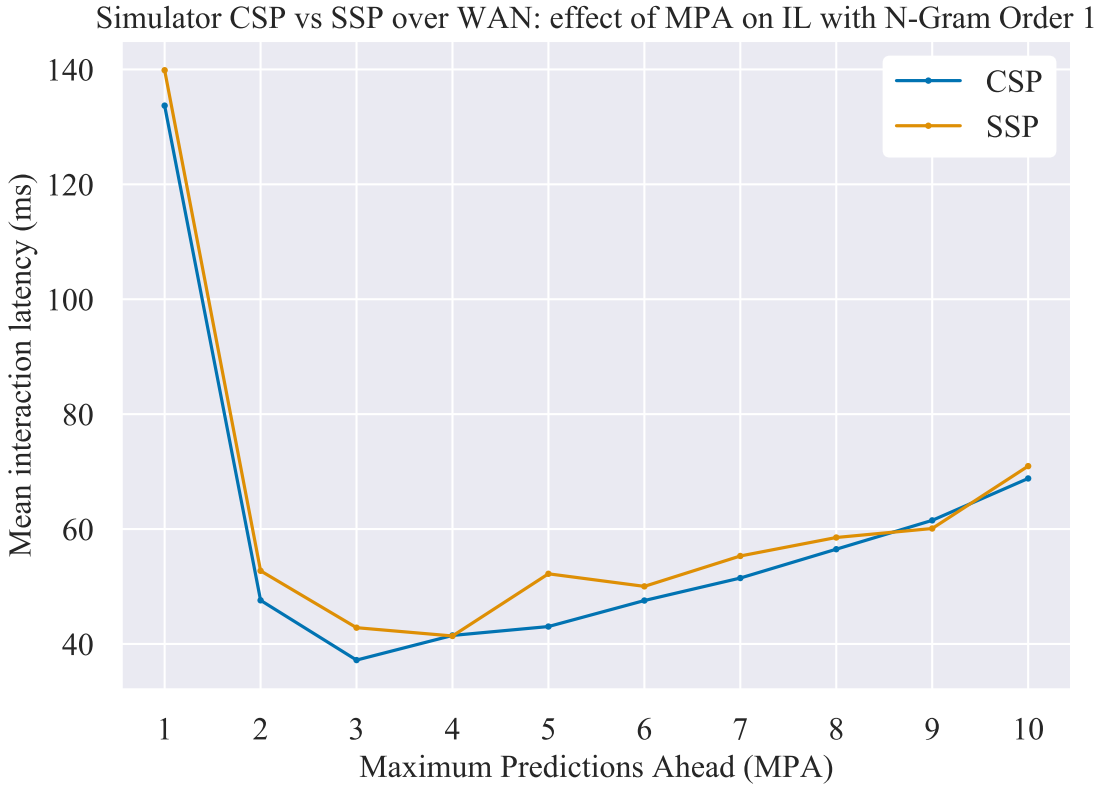


Figure 9.14 Simulator CSP vs SSP over WAN comparison.

Table 9.6 This table shows the mean IL difference ($\Delta\overline{IL}$) between CSP and SSP modes using the simulator and running over WAN.

Order	1	2	3	4	5	6	7	8	9	10
$\Delta\overline{IL}$	6.16	5.14	5.64	0.09	9.18	2.47	3.84	2.04	1.42	2.15
Configuration with lowest IL	CSP	CSP	CSP	SSP	CSP	CSP	CSP	CSP	SSP	CSP

In Figure 9.14 and Table 9.6, the results for the experiments performed with the configuration described in Table 9.5 Groups *A* and *B* are presented. However, only WAN was used in these experiments. As can be seen in Figure 9.14, both CSP and SSP, IL falls from MPA = 1 until a certain point: MPA reaches the ideal value for the NL being experienced. Once reaching the ideal MPA, IL beings to gradually climb, which is observed, again, in both CSP and SSP. Interestingly, though, CSP appears to provide lower IL. This is contrast with the results using the simulated NL where SSP is shown to yield lower IL. Unfortunately, WAN conditions are not stable, and ping fluctuates (sometimes with latency spikes of 10s of ms) and is therefore likely responsible for the contradictory measurements.

9.4.2. Unity3D CSP vs SSP

Using the configuration parameters described in Table 9.5 Groups *C* and *D*, 700 experiments were performed, the results of which are presented below in Figure 9.15.

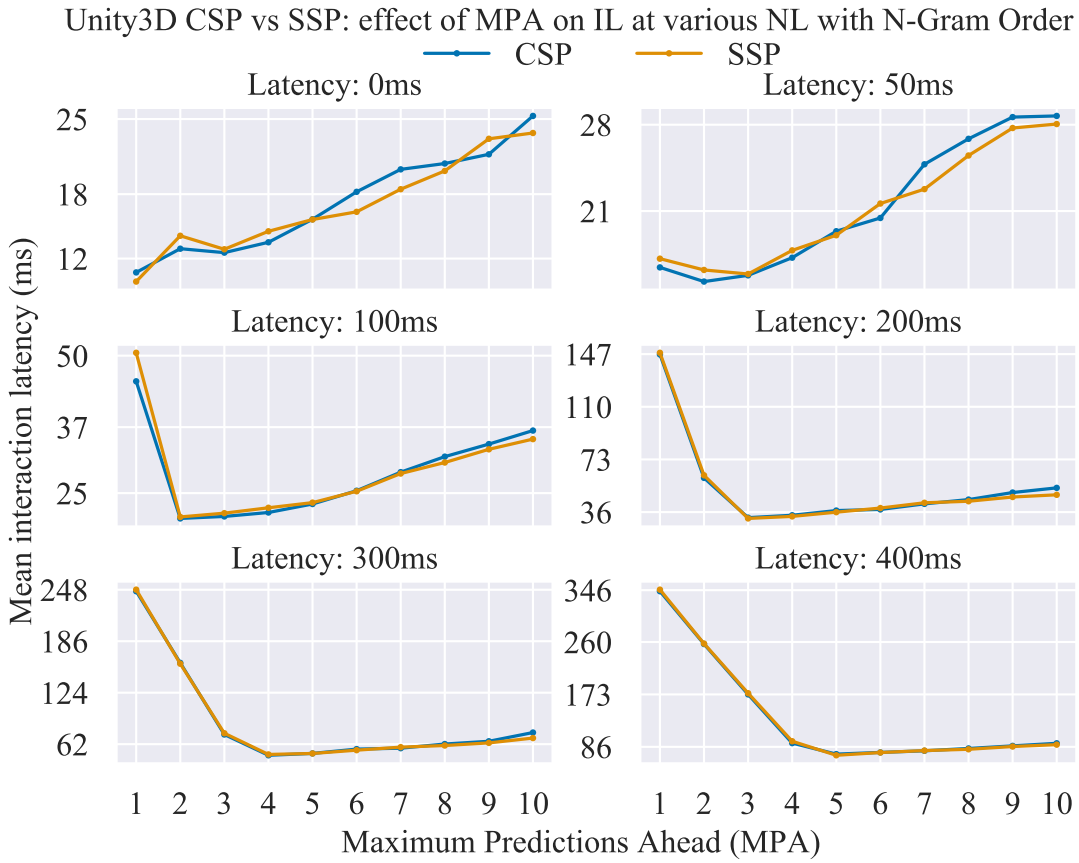


Figure 9.15 A comparison of Unity CSP vs SSP. MPA.

Figure 9.15 shows that mean IL decreases with MPA until an ideal value is reached, at which point, mean IL begins to climb. This is true for all experiments evaluating MPA. Like measurements comparing CSP and SSP for the simulator, the CSP mode IL results are higher than those taken using SSP for the Unity3D PIRR system. When inspecting $\Delta\overline{IL}$, the simulator does not show clearly the same increase over MPA.

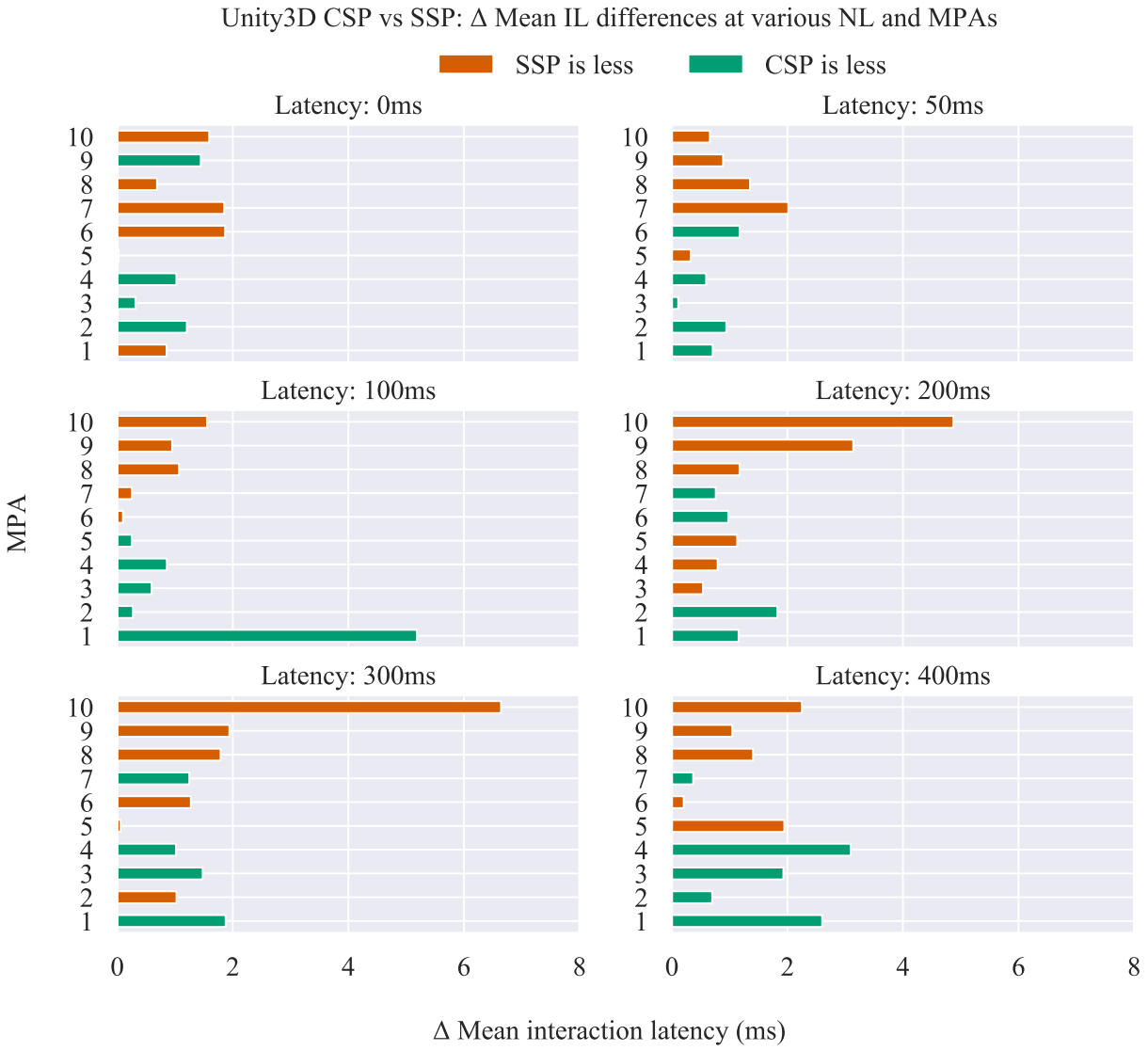


Figure 9.16 Unity CSP vs SSP. Δ Mean IL differences at various NL and MPAs

9.4.3. Simulator CSP vs Unity3D CSP

In all cases thus far, SSP has been shown to yield slightly lower mean IL, particularly at higher MPA levels; this was demonstrated for both the simulator and Unity3D platform. However, in order to understand how closely the simulator mimics the Unity3D PIRR system, it is necessary to compare the simulator CSP with that of Unity3D; this comparison is illustrated in Figure 9.17. No additional experiments were performed here and instead, the results from previous runs are used.

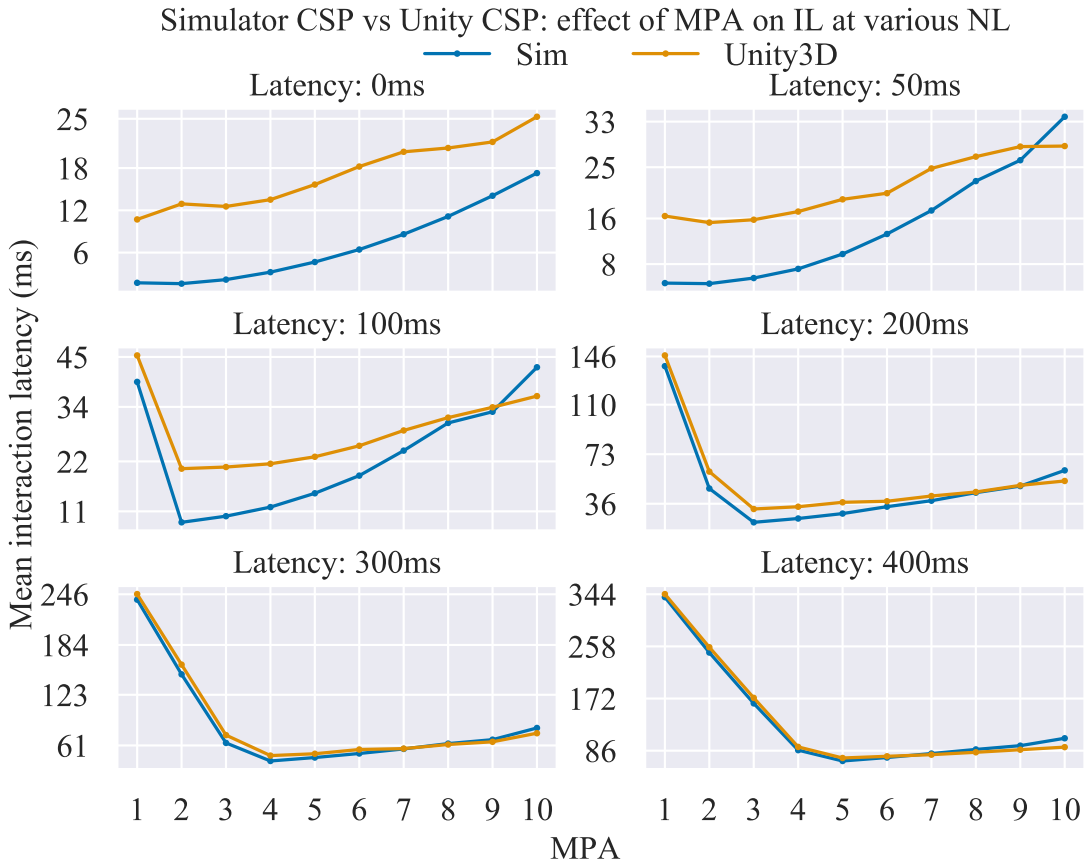


Figure 9.17 Simulator vs Unity3D CSP: Mean IL various NL and MPA values.

From the figure, Unity3D tends to provide higher mean IL at lower latencies and at smaller MPA values. As NL and MPA increase, however, the Unity3D PIRR system and the simulator produce more comparable results and at higher MPA values, the Unity3D platform tends to produce lower IL.

Figure 9.18 Simulator vs Unity3D CSP: Δ Mean IL at various NL and MPA values.

The mean difference between the simulator and Unity3D platforms were then calculated across MPA levels for each NL, and the results plotted in Figure 9.18, which shows that in most cases, the simulator produced lower mean IL. Why the simulator produced lower mean IL is not clear, however it is likely due to the simulator having less variability than Unity3D, which itself is likely caused by rendering delays and background processes of Unity3D.

9.4.4. Simulator SSP vs Unity3D SSP

In this section, SSP modes were compared between the simulator and Unity3D platforms. No new experiments were performed; rather, previously recorded results were used to identify the differences between the platforms running in SSP mode. The results illustrated in Figure 9.19 show nearly identical plots to those of Figure 9.17.

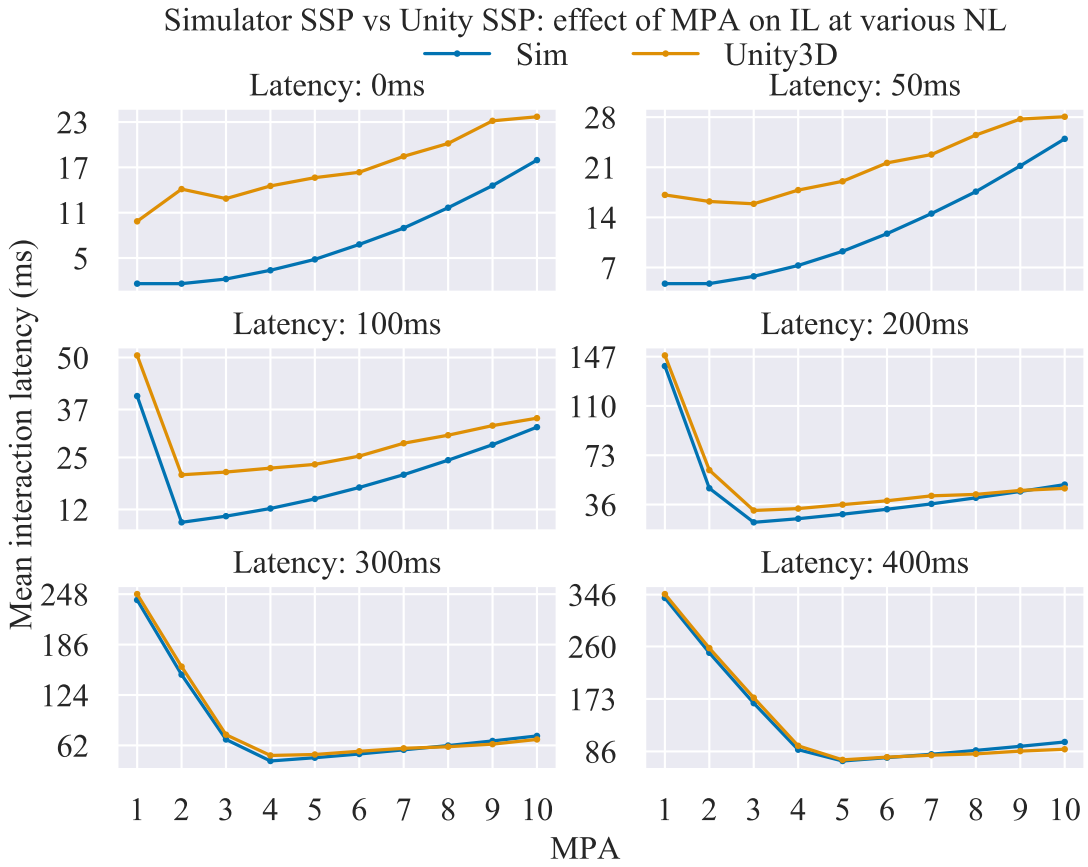


Figure 9.19 Simulator vs Unity SSP. Mean IL measured at simulated NL's. MPA.

As can be seen from the figure, the simulator and Unity3D platforms produce similar mean IL across all simulated NL. Additionally, like with the CSP comparison in Figure 9.17, IL generated by Unity3D is higher at lower NL values. However, as NL and MPA increase, the IL experienced by Unity3D gradually decreases until it is lower than that produced by the simulator. Looking at the $\Delta\overline{IL}$ values in Figure 9.20, the simulator consistently produces lower mean IL at all MPA values, until NL is increased to 200ms. At this point, the Unity3D begins to result in lower mean IL, but only at higher MPA values. This indicates that if MPA or NL were to increase, the unity3D platform would continue to provide lower mean IL. In all, these results indicate that while the simulator is able to behave in a comparative manner at lower NL, a discrepancy between the simulator and Unity3D does grow at higher NL and as such, would need investigating before considering measurements at higher NL reliable.

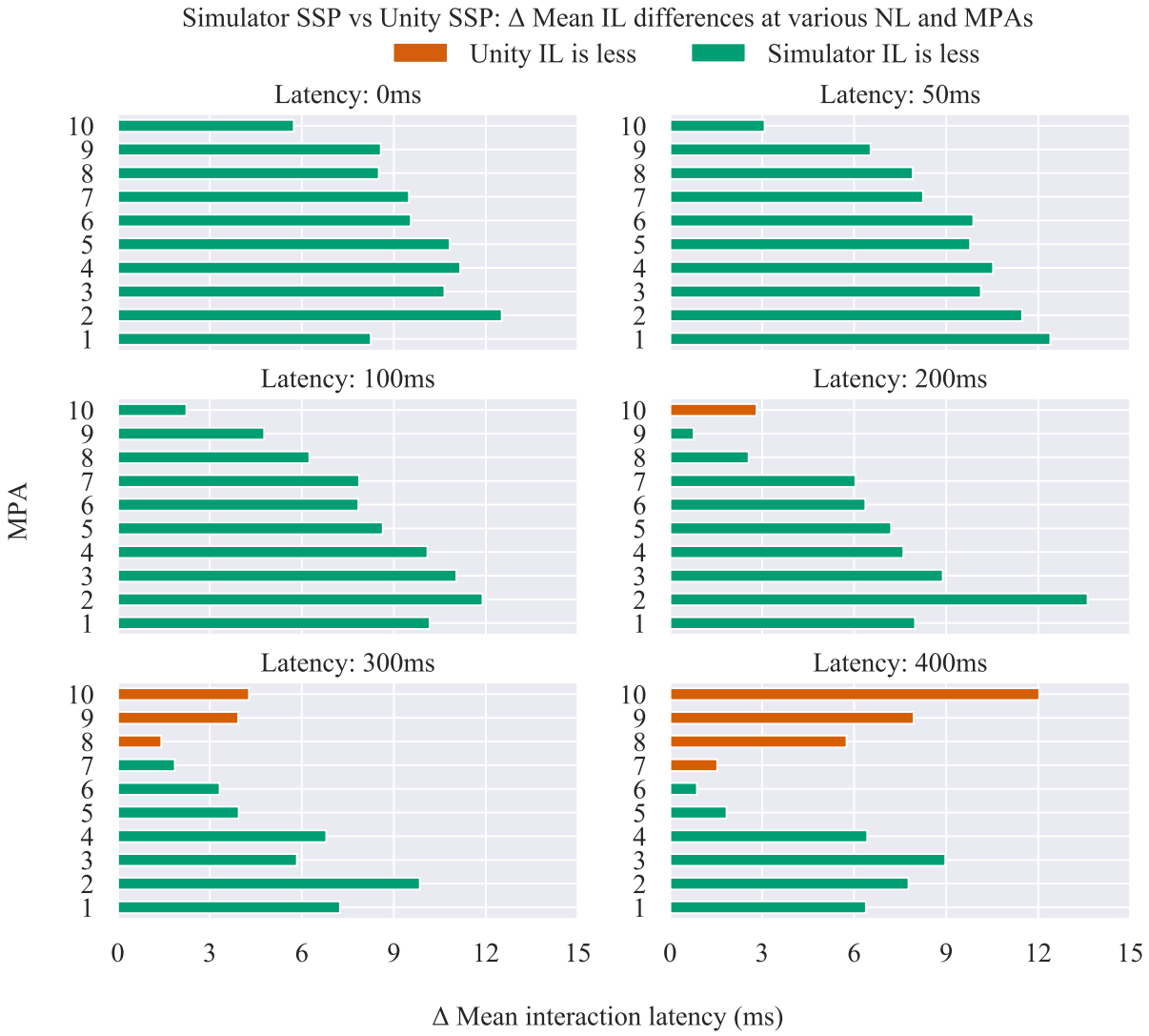


Figure 9.20 Unity vs simulator SSP. Δ Mean IL differences at various NL and MPAs.

9.4.5. Summary

In this chapter, CSP and SSP configurations are described, and how they are implemented in both the simulator and Unity3D PIRR systems is explained. The focus of this chapter was to determine if there is any noticeable difference between CSP and SSP configurations, as well as between the simulator and Unity3D platforms when operating in these modes. CSP and SSP configurations were therefore examined with respect to the effect of N-Gram Orders as well as MPA's on IL. In addition, the simulator and Unity3D platforms were compared.

In the first group of experiments involving N-Grams (see in §9.3), CSP was found to provide lower mean IL in all cases, however, the differences are negligible. While CSP vs SSP differences are shown to be very low with a maximum of approximately 5ms, the simulator and Unity3D comparisons show larger discrepancies. For example, in the CSP experiments comparing the two platforms, the largest difference was found to be about 12ms at NL = 0ms, which decreased to less than 1ms at NL = 400ms. This is in contradiction to the same

comparison for SSP, where the differences were seen to increase with NL, starting at approximately 1ms for NL = 0ms, and climbing to nearly 30ms for NL = 400ms.

In terms of MPA (see §9.4), SSP was shown to yield lower mean IL in many cases, but this mostly occurred at higher MPA levels. The advantage shown by SSP mode is likely due to the prediction module not impacting the client device in terms of compute resource, compared with CSP.

When comparing the simulator and Unity3D platforms with regards to CSP and SSP modes, the simulator consistently yielded lower IL at lower MPA levels as well as at lower NL. However, as MPA and NL increases, the difference between the two platforms diminishes; this can be seen in Figure 9.18 and Figure 9.20. The exact cause of these observations is not clear, but it is likely due to an issue with the simulator: thread synchronisation or perhaps CPU time slicing causes the simulator to lose a few milliseconds per experiment.

Chapter 10. Discussion and Conclusions

10.1. Discussion

Chapter 3 examined the modelling, simulation and measuring of latency in IRR systems. Latency points were mapped out and the main sources of latency were identified. Techniques to measure IL were then introduced and, given the flexibility in being able to measure at various locations in the IRR pipeline, the integrated approach was selected as the most suited for IRR systems. In order to experiment with latency measurement, a simple client-server IRR system was developed with Unity3D. However, it was soon identified that measuring over WAN is not reliable since latency is outside of the operators control and therefore will prevent the ability for reproducible experiments to be performed. As such, a latency simulator was developed and after evaluation it was found to behave reasonably well when compared with WAN experiments. In all experiments measuring IL using the integrated approach, the mean IL was found to be $\approx$ 16ms to 18ms (Figure 3.9) above the introduced latency. This was the expected result as the IRR system had a similar mean baseline. The experiment using WAN and the integrated approach again confirmed that the latency simulator operates well (Figure 3.9), with a mean difference of just 0.68ms – a negligible amount considering a system latency of 170ms. It is clear, though, that there is noise in the results. However, it is suspected the noise is due to the 15ms time-slice limitation of the Windows OS, as well as the challenges of multi-threading applications and the lack of control over thread execution.

The integrated approach to latency measurement was described and explored and found to be suitable for measuring IL in IRR systems, but required being tightly woven into the codebase, making it unsuitable for general use. The approach is robust in that the developer has complete control over where measurements are performed and can collect measurements at various points in the pipeline. However, in the case where source code access is not available, the hardware approach is a suitable alternative so long as the correct hardware (high-speed camera, tracker, LED, etc.) is available to the operator and care is taken to set up and perform measurements accurately. The observer approach (such as in the example of Chen; described in §3.4.1), is useful but has a number of issues and lacks the ability to consider measurements of applications consisting of moving entities which are not coupled to user interaction (e.g. grass blowing in the wind). This approach was expanded on by capturing screen pixels (by reading from the framebuffer) within a user-controlled reticule and by monitoring PSNR values for the identification of interaction results arriving from the server.

In general, latency measurement can be challenging to preform, especially if there is no access to the source code of the application. Therefore, a new approach was developed and is referred to as the Latency Measurement Tool (LMT) in this research. It was found that by recording both the application of interests window and user interactions and timestamping both activities, the Peak Signal to Noise Ratio can be used to detect interactions from captured frames, whose timestamps can be cross-referenced with logged interactions and therefore enables the measurement of IL. The new tool is able to measure IL for virtually any application, whether operating over a network or not. It was evaluated against the integrated mode and found to be comparable, but a comparison with other approaches such as the hardware technique will further verify this new tool. In addition, it was hoped that the tool would be suitable for measuring IL in PIRR systems, but unfortunately the measurement process is still rather manual requiring the operator to physically initiate capturing and then perform interactions and therefore is not yet ready for use in automated experiments. However, it should be possible to automate this process. Further, there are some notable limitations. For example, the LMT developed has mean capture time ($\overline{C_t}$) dictated by the monitor refresh rate (60Hz), which in the described case was $\approx$ 16ms. There is therefore a risk that a measurement will be out by at least $\overline{C_t}$ (if pixel change occurs before or after the capture). Nevertheless, the results when comparing the LMT with the integrated approach indicate that the tool does indeed capture IL with reasonable accuracy. Interestingly, the fact that the LMT captures a consistent amount of latency more than the integrated approach suggests that it may be capable of capturing latency which the integrated approach is not. Nevertheless, more exploration and perhaps a comparison with the hardware approach is required.

Chapter 4 looked at developing an N-Gram model so that keyboard user interactions could be predicted. Only keyboard interactions were chosen for modelling, because the focus of this thesis was on developing a framework for using prediction in IRR systems and evaluating its effect on IL, meaning that a comprehensive approach to user interaction prediction is not required. Initially, a buffer-based approach to the storage and retrieval of predictions was implemented. This was found to perform well initially, but with usage, it became clear that latency was building up between predictions. The issue was the growing size of the buffer and the increase in search time. Therefore, a dictionary-based data structure was developed and introduced, solving the issue of latency growth.

Chapter 5 aimed to develop an IRR simulator framework with prediction (PIRR). To this end, a client and server application were created, as well as a prediction module which incorporated the model described in Chapter 3. Additionally, the latency simulator from Chapter 2 was packaged into a module and integrated into the simulator in a CSP configuration.

RabbitMQ was used for communications between the different applications. TCP was initially implemented manually but after encountering issues such as managing multiple simultaneous

connections, it was decided that a ready-made, "battle-tested" solution such as RabbitMQ was chosen. RabbitMQ was selected over other options due to its easy integration with Unity3D as well as its C# bindings. This choice made it easy to experiment with different configurations, CSP and SSP modes, as well as with the multi-track prediction described in Chapter 8. In hindsight, ZeroMQ would likely have been a better choice of messaging library since it is well known for its unmatched performance [89]. UDP was not considered, despite being a faster protocol, because of the mandatory requirements that system messages must arrive at and be sent from the relative applications in the order in which they are sent, and without fail. In hindsight, had everything been developed natively with TCP, bandwidth usage could have been monitored and reported – something this research lacks.

A single-track prediction scheme was implemented with the N-Gram model allowing predictions along a single trajectory to be made, with the prediction module taking advantage of network delay between the client application and itself. It was found that integration of the prediction model was not straightforward, since in order to perform IL measurements using the integrated approach, there needs to be a link between actions performed/simulated on the client application, predictions generated, and the results produced by the server application. Further, some prediction results may arrive on the client before any corresponding action has been performed/simulated and as a result, these "future" predictions need to be stored for possible use at a later time. As such, a special "key" was devised for ensuring that prediction results could easily be identified as expired or whether being required for the future. In addition, the unique key allows for prediction results to be matched with actions arising on the client application and makes it easy to identify whether the prediction result is correct or not.

In Chapter 6, the simulator was used to perform experimentation with prediction and IL. The primary goal of this chapter was to determine whether or not prediction had any effect on IL and if it did:

1. What is the effect of prediction on IL?
2. How much prediction "history" should be used in performing predictions?
3. What is the effect of using more prediction history?
4. How far ahead should predictions be performed at various latencies?
5. How can system recovery times (increased due to incorrect predictions) be calculated?

Various latencies were chosen for simulation and WAN was also used for experimentation. The results indicate that the immediate last interaction performed is the best indicator of the next likely prediction: IL is lowest when N-Gram Order = 1 and this is true for both simulated latencies and WAN. Additionally, it was found that how far ahead to predict is dependent upon network latency as well as the rate at which interactions are performed or simulated. MPA was

also found to cause issues when too high, since valuable render time is blocked. To measure system recovery time after an incorrect prediction, local maxima (peaks in the IL signal) where located and the delay between peak base points was calculated. With this method, it was found that recovery times increased with N-Gram Order as well as with MPA, indicating that an incorrectly chosen MPA has more of an effect (negative) on IL than on an incorrectly chosen N-Gram Order.

In Chapter 7, a real-world PIRR system was built with Unity3D. A game engine was chosen due to the rendering, interaction, compression and scene building capabilities; this saved a lot of development time. The development of the Unity3D PIRR system aided, in part, the design if the simulator platform. Indeed, the two platforms were developed concurrently. This was mostly due to the way in which interaction is managed by a games engine and the intention to have the simulator mimic the real-world one as closely as possible. Nevertheless, complications arose with the introduction of the prediction module. For instance, physics had to be disabled so that prediction would be permitted to produce results well ahead of the present position of the player avatar, which resulted in the loss of ability for collision detection between scene objects and caused the avatar to pass through walls and other objects. This was fixed by transmitting to the client a list of bounds for which the user is not allowed to cross. However, this solution does break the aim of image-based PIRR systems in that pure image-only results were no longer being transmitted. This is not of serious concern, however, since the data is not believed to be sufficient to recreate any models, nor is it thought that collision detection is used in data visualization. Further complications arose with scene background objects and the current system is not suitable for such objects. This is because when predicting ahead, results quickly become out of date which limits severely how far ahead usable prediction can be made. It is thought that this issue can be addressed by building into the rendering system a manageable state where the scene can be “rewound” to an earlier point in time. If the rendering engine was able to produce multiple renders simultaneously, valuable render time would not be blocked, especially when mispredictions occur and that time is wasted. Unfortunately, Unity3D does not allow this operation.

Once the system had been developed, experimentation began. The same experiments conducted with the simulator were performed with the Untiy3D system and results were compared. It was found that base operating IL measurements were comparable to the simulator, both when using simulated latencies as well as over WAN, with results often indicating less than 1ms difference. However, the system did produce more variable data than the simulator. This is due to the nature of the rendering engine and the fact that rendering delay is not controlled and therefore fluctuates quite often. This variance is clearly presented by Figure 7.7. A potential improvement would be to make the server application simulate rendering by waiting random amounts of time according to the distribution of delays produced by the real renderer. When prediction is enabled, it is clear that it has as positive effect on IL and like the simulator, IL increased with

N-Gram Order and an Order of 1 was found to be the most suited. Similarly, it was found that the Unity3D system exabits the same characteristics as the simulator in terms of MPA: IL decreases up until a point (depending on NL) with MPA increase, but then begins to climb. However, the results produced by this platform produce significantly higher values indicating that the effect of MPA is different on the Unity3D system than it is on the simulator; more investigation is required to determine the source of this discrepancy.

In Chapter 8, CSP and SSP were introduced. The results gathered from the simulator and Unity3D platforms were reused and their results were compared in both CSP and SSP modes. CSP and SSP for individual platforms was also investigated. For N-Grams, it was found that in all cases and regardless of what platform was used, CSP yielded lower IL results. It is thought that this is due to fact that in CSP mode, the prediction module is "closer" to the client application, which allows for incorrect predictions to be detected sooner than in SSP mode. In terms of MPA, SSP was found to be more beneficial for IL at higher MPA values. At lower levels, both CSP and SSP produce similar results, with CSP sometime producing lower IL. However, as NL increases and as a higher MPA is required to mask the incurred latency, SSP begins to demonstrate lower IL measurements. Why this happens is not exactly clear, but it is thought that the client application performs better when the computer on which it is operating is unburdened by the additional processing requirements of the prediction module. Before either CSP or SSP can be declared as "better" for IL, more work is required, however. For example, this research has not investigated the scalability of the system, nor of the potential for ensemble prediction models or more computationally expensive once, both of which will consume more resources than the proposed approach and likely impact the client application further.

Chapter 9 investigated how PIRR system mispredictions can be managed. Various approaches were described but ultimately, the presented multi-track approach introduced as a possible solution. During early stages of experimentation, it was found that a single instance of the server application was not sufficient for the number of prediction messages being generated and needing to be processed. As a response, the PIRR architecture was modified to allow for multiple server application instances. While initial experiments looked promising, successfully eliminating the IL cost incurred from misprediction and resulting in zero IL when the system was run with no introduced simulated NL, later experiments found complications. For instance, despite distributing the predictions to be processed over a number of server applications, the client application was unable to process the large influx of server responses fast enough to prevent a backlog from forming. Further, it was identified that a maximum of MPA = 3 could be used, since the client application is unable to cope with a higher number. It was also identified that between 6 and 7 server applications is required in order to prevent a backlog forming on the server. All together, these results indicate that multi-track prediction is unlikely to be a viable solution to misprediction compensation, and a new approach is required. The only situation in

which this scheme may be useful is when rendering of some predictions can be skipped since previously visited areas have already been processed.

10.2. Conclusions

The aim of this research was to explore the measurement, modelling and management of IL in IRR systems, with a focus on the following key questions: What is the effect of prediction on IL? How can we model and simulate latency? How can we measure IL when prediction is used? What is the optimal number of predictions ahead required to minimize latency? On which side of the network should prediction be performed?

In an attempt to answer the research questions, set out in this work, additional ones were raised, resulting in a set of questions describing an end-to-end solution for the simulation and construction of prediction-enabled IRR systems. The following section addresses these additional questions as well as the main thesis ones.

10.2.1. Research questions

RQ1. How can IL be modelled, measured and simulated?

Latency is present in all distributed systems and IRR systems are no exception. It is therefore important to be able to model and simulate this latency so that experiments can be performed in controlled and reproducible conditions. Due to the variability and not-in-our-control nature of WAN's, they are not suitable for experiments in which fine-grained control over latency is required. How to model and simulate latency was addressed by first constructing a latency model and then by developing a latency simulator capable of being integrated into any IRR system components. The latency simulator was validated and found to be capable of simulating latency comparable to that of real world WAN. Before validating, a method for consistently and accurately measuring IL had to be addressed, which is answered in the next question.

To measure IL, the latency model developed was used to identify key IL measurement locations. An integrated approach to IL measurement was introduced and then built into a simple client-server application using Unity3D. It was found that in order to perform measurements, it is necessary to correctly identify which arriving results correspond with the performed interactions. This was overcome with a simple stopwatch and ID approach. With the ability to simulate and measure IL, validation on the simulator was performed (Figure 3.9) and it was found that the difference in IL between measurements collected when using a live network and those of a simulated one were less than 1ms on average. It was identified that there are no generic, easy to use IL measurement tools available and those that do exist, require unreal expectations of general users wishing to perform measurement themselves. As such, a novel software-based approach to measurement was introduced and validated, demonstrating a robust

method for measuring IL suitable for a range of applications. This new approach was validated using a simple Windows application and then on the earlier used client-server application.

RQ2. Are N-Grams suitable for keyboard-based user interaction prediction?

An N-Gram model was developed and tested in order to predict keyboard-based user interactions. The idea is that if interactions can be predicted, corresponding rendering can be performed in advance (on a remote server), and the render results can be delievered to a client application ahead of time. Correctly predicted results can then be displayed to the user at the time of interaction and therefore mask the delays introduced by a network.
An initial implementation used a simple buffer to store predictions, but this approach was found to be flawed, resulting in an ever-increasing prediction computation time. As such, a new dictionary-based model was developed which produced predictions with less than a millisecond of computation delay. The model was evaluated in trained and untrained modes, and it was found that accuracy was approximately 83%, regardless of whether or not the model was trained beforehand. It was also identified that the very last interaction is the best indicator of the immediate next interaction.

RQ3. How can prediction be integrated into an IRR system and what is the impact ofprediction on IL?

To answer this question, an IRR simulator platform with an integrated prediction module was developed. This was done so that a experiments could take place in a more controllable environment and so that the system was “open”, without any constraints or limitations potentially imposed by using a real rendering engine. Results showed that using prediction does indeed lower IL, but the introduction of prediction complicates the process of performing IL measurements, and ultimately required a modification to the integrated IL measurement approach. A real-world IRR system using Unity3D was then developed and the prediction module from the simulator was integrated. The IRR system provided insight into some of the remaining issues this topic faces. For example, the current implementation does not allow for background scenery to move independently of user interaction due to the state-like approach taken. Another issue found was that incorrect predictions cause a visible delay to the system, raising the question of how to manage incorrect predictions, which is discussed in Chapter 8. Finally, the use of a games engine did, in addition to the increase in productivity, provide a valuable means for validating the simulator platform, which has been shown to be representative of a real-world system.

Should predictions be generated on the client or on the remote server?

During experimentation it was identified that the prediction module could be placed either on the same side of the network as the client application, or on that of the server application. A final contribution that this work makes is that of the investigation of CSP and SSP modes.

Experimentation found that with both the simulator and Unity3D platforms, CSP is always lower than SSP, irrespective of how many historical interactions are used for prediction. In terms of MPA, it was found that SSP is beneficial for larger MPA levels than CSP and that CSP is more suitable for lower MPA levels.

In conclusion, this thesis contributes to the understanding of how to model, simulate and measure interaction latency; introduces a PIRR simulator platform; describes how to predict and integrate keyboard-based interaction prediction into PIRR systems; explores how far head to predict and how much history to use; evaluates the effect of prediction on IL and which side of the network prediction should be made; and describes how to calculate the time IL takes to return to normal within PIRR systems after an incorrect prediction is experienced.

The proposed methods may help towards the challenges presented by IL, especially for application such as Google Street View, where the 3D environment is static. The methods for measuring IL are generic and are not specific to a certain configuration or system environment. Further, the approach described to calculate the recovery time of PIRR systems following an incorrect prediction can be applied to generalised to any system in which the duration of abnormal events need to be calculated. Although the results presented here are not yet ready for use in production systems and more research is required, the work helps to lay the foundations of prediction in future PIRR systems and provides new tools and methods to assist future exploration of these systems.

10.3. Future work

10.3.1. Real-time Operating Systems

IL in the IRR system was measured using the integrated approach and while the obtained measurements are reliable, there is always noise present in IRR systems (e.g. rendering delays) and the system described is no exception. For example, in Figure 3.6 and others, there is a large amount of variability in the data. This suggests that there is an underlying process which is causing instability in the IRR system. The most likely cause is the chosen method for "pausing" a thread. To do this, the standard .NET Thread.Sleep() function was used. Unfortunately, Windows-based machines are not real-time operating systems (OS), providing a thread time-slice of 15ms. As a result, the sleep function has a resolution of 15ms. It has been noted that the time-slice can be configured to 1ms by using timeBeginPeriod and timeEndPeriod, however, there was reluctance to do this as it would affect the thread time-slice OS-wide and may have unexpected consequences. While the apparent system noise is likely of little concern, it would interesting reproduce and further verify the introduced approach with a real-time OS.

Finally, while the use of a 3D rendering engine such as Unity3D contributes significantly towards the rapid development of IRR systems, the introduction of prediction results in some

severe limitations. The first and most critical issue is that rendering engines are generally single-threaded, meaning that there is a single "render loop" and that all data transformations and procedures must eventually execute on this "main" thread. In PIRR systems, the ability to render multiple predictions simultaneously would be extremely advantageous, since unfortunately, it is not possible to identify incorrect predictions until the client has either received the incorrect result or the prediction module has received an update from the client, at which point incorrect predictions will be have been (or be in the process of) rendered, blocking the rendering engine and wasting critical milliseconds which ultimately impacts IL.

10.3.2. Context Awareness

Context-aware prediction models may be useful in predicting not only what interaction will be performed, but also when that interaction will occur. For example, in situations where a user responds to onscreen stimuli, a context-aware prediction model may be able to assist in predicting when an interaction is about to be performed by monitoring the distance of a player to a wall and providing probabilities as to how likely the user is to take action given historical situations. Similarly, context-aware models could monitor the PSNR between frames and detect significant scene changes in simulations of time-varying data, potentially helping the model in identifying the likeliness of an interaction event. The type of interaction to be performed can also be narrowed-down with context-aware prediction models, since certain events can be filtered out. For instance, in returning to the VE wall example, moving forward could be filtered out as it is more likely that the user would want to either change direction or stop than walk into the wall.

10.3.3. Foveated Rendering

Foveated rendering is a state-of-the-art graphics rendering technique which aims to use eye-tracking technology to determine what a user is looking at within a scene. Using that information, content laying within the peripheral vision of the user is rendered at much reduced image quality, while the part of the scene the user is focused on is rendered in full quality. The result is that a significant portion of the number of pixels required to be rendered is reduced. In fact, a study in 2016 [90] demonstrated that this degradation of peripheral pixels is invisible to users. At present, the focus of this is primarily aimed at Virtual and Augmented reality technologies. However, this same approach may prove beneficial for reducing IL in PIRR systems and therefore is a promising direction for future research.

Appendix A. Measuring recovery times

```
from scipy.signal import find_peaks
import numpy as np
import pandas as pd

def get_mean_recovery_period(data):
    il = data['interaction_latency']
    peaks, _ = find_peaks(il, height= np.mean(il))
    peaks = np.insert(peaks, 0, 0)
    index_tuples = get_recovery_period_indices(peaks, il)
    results = []

    for start, end in index_tuples:
        delay = data.iloc[end]['run_time'] - data.iloc[start]['run_time']
        results.append(delay)

    return np.array(results).mean()

def get_adjacent_peaks(peaks):

    results = []

    for i, peak in enumerate(peaks):
        if i < len(peaks)-1:
            if i == 0:
                left_peak = 0
                right_peak = peaks[1]
            else:
                left_peak = peaks[i-1]
                right_peak = peaks[i+1]

            results.append({'left': left_peak, 'mid': peak, 'right': right_peak})

    return results
```

```
def get_recovery_period_indices(peaks, data):

    data =data.values

    parts = get_adjacent_peaks(peaks)
    results = []

    for i, part in enumerate(parts):

        left = part['left']
        mid = part['mid']
        right = part['right']

        left_il = data[left: mid]
        right_il = data[mid: right]

        if len(left_il) == 0: left_mean = 0
        else: left_mean = np.array(left_il).mean()
        right_mean = np.array(right_il).mean()

        left_idx = 0
        right_idx = 0

        for x in range(len(left_il), 0, -1):
            if len(left_il) > 0:
                value = left_il[x-1]
                if value > left_mean:
                    left_idx = x-1
                else:
                    left_idx = left + x
                    break

        for y in range(0, len(right_il)):
            if len(right_il) > 0:
                value = right_il[y]
                right_idx = mid + y
                if value > right_mean:
                    pass
                else:
                    break

        tup = (left_idx,right_idx)
        results.append(tup)
```

```
    return results
```

References

[1] R. Jota, A. Ng, P. Dietz, and D. Wigdor, "How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '13, (New York, NY, USA), p. 2291–2300, Association for Computing Machinery, 2013.

[2] M. Claypool, K. Claypool, and F. Damaa, "The effects of frame rate and resolution on users playing first person shooter games," in Multimedia Computing and Networking 2006 (S. Chandra and C. Griwodz, eds.), vol. 6071, pp. 1 – 11, International Society for Optics and Photonics, SPIE, 2006.

[3] R. S. Allison, L. R. Harris, M. Jenkin, U. Jasiobedzka, and J. E. Zacher, "Tolerance of temporal delay in virtual environments," in Proceedings IEEE Virtual Reality 2001, pp. 247–254, March 2001.

[4] J.-R. Wu and M. Ouhyoung, "Reducing the latency in head-mounted displays by a novel prediction method using grey system theory," Computer Graphics Forum, vol. 13, no. 3, pp. 503–512, 1994.

[5] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," IEEE Transactions on Visualization and Computer Graphics, vol. 20, pp. 2122–2131, Dec 2014.

[6] N. Holliman and P. Watson, "Scalable real-time visualization using the cloud," IEEE Cloud Computing, vol. 2, pp. 90–96, Nov 2015.

[7] M. Šterk, M. Agustín, and C. Palacio, "Virtual globe on the android - remote vs. local rendering," ITNG 2009 - 6th International Conference on Information Technology: New Generations, p. 634–639, 2009.

[8] F. W. B. Li, R. W. H. Lau, D. Kilis, and L. W. F. Li, "Game-on-demand:: An online game engine based on geometry streaming," ACM Trans. Multimedia Comput. Commun. Appl., vol. 7, Sept. 2011.

[9] S. Shi, K. Nahrstedt, and R. Campbell, "A real-time remote rendering system for interactive mobile graphics," ACM Trans. Multimedia Comput. Commun. Appl., vol. 8, Oct. 2012.

[10] J. Doellner, B. Hagedorn, and J. Klimke, "Server-based rendering of large 3d scenes for mobile devices using g-buffer cube maps," in Proceedings of the 17th International Conference on 3D Web Technology, Web3D '12, (New York, NY, USA), p. 97–100, Association for Computing Machinery, 2012.

[11] F. Lamberti and A. Sanna, "A streaming-based solution for remote visualization of 3d graphics on mobile devices," IEEE Transactions on Visualization and Computer Graphics, vol. 13, pp. 247–260, March 2007.

[12] C.-H. Chu, Y.-H. Chan, and P. H. Wu, "3d streaming based on multi-lod models for networked collaborative design," Computers in Industry, vol. 59, no. 9, pp. 863 – 872, 2008. S.I. Advances in Collaborative Engineering: From Concurrent.

[13] A. Wessels, M. Purvis, J. Jackson, and S. Rahman, "Remote data visualization through websockets," in 2011 Eighth International Conference on Information Technology: New Generations, pp. 1050–1051, April 2011.

[14] I. E. Sutherland, "Sketch pad a man-machine graphical communication system," in Proceedings of the SHARE Design Automation Workshop, DAC '64, (New York, NY, USA), p. 6.329–6.346, Association for Computing Machinery, 1964.

[15] C. Wylie, G. Romney, D. Evans, and A. Erdahl, "Half-tone perspective drawings by computer," in Proceedings of the November 14-16, 1967, Fall Joint Computer Conference, AFIPS '67 (Fall), (New York, NY, USA), p. 49–58, Association for Computing Machinery, 1967.

[16] "First rendering: Very first complex 3d rendered object now rendered in html5." https://firstrender.net. online.

[17] "Our story." https://www.pixar.com/our-story-pixar. online.

[18] D. A. Price, The Pixar touch: the making of a company. Vintage Books, 2009.

[19] M. Oppitz and P. Tomsu, Inventing the Cloud Century: How Cloudiness Keeps Changing Our Life, Economy and Technology. Springer International Publishing, 2018.

[20] "A brief history of cloud computing." https://www.ibm.com/blogs/cloud-computing/2014/03/18/a-brief-history-of-cloud-computing-3/, Aug 2019. online.

[21] J. Ahrens, S. Jourdain, P. O'Leary, J. Patchett, D. H. Rogers, and M. Petersen, "An image-based approach to extreme scale in situ visualization and analysis," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, p. 424–434, IEEE Press, 2014.

[22] A. Boukerche and R. Pazzi, "Scheduling and buffering mechanisms for remote rendering streaming in virtual walkthrough class of applications," pp. 53–60, 01 2006.

[23] A. Boukerche, R. Werner, and N. Pazzi, "A peer-to-peer approach for remote rendering and image streaming in walkthrough applications," IEEE International Conference on Communications, p. 1692–1697, 2007.

[24] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn, "Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming," in Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '15, (New York, NY, USA), p. 151–165, Association for Computing Machinery, 2015.

[25] C.-H. Chu, Y.-H. Chan, and P. H. Wu, "3d streaming based on multi-lod models for networked collaborative design," Computers in Industry, vol. 59, no. 9, pp. 863 – 872, 2008. S.I. Advances in Collaborative Engineering: From Concurrent.

[26] J. Park and H. Lee, "A hierarchical framework for large 3d mesh streaming on mobile systems," Multimedia Tools and Applications, vol. 75, pp. 1983–2004, Feb 2016.

[27] Z. Tang, O. Ozbek, and X. Guo, "Real-time 3d interaction with deformable model on mobile devices," pp. 1009–1012, 11 2011.

[28] C. D. Hansen and C. R. Johnson, The visualization handbook. Elsevier-Butterworth Heinemann, 2005.

[29] S.-C. Cheng, C.-T. Kuo, and D.-C. Wu, "A novel 3d mesh compression using mesh segmentation with multiple principal plane analysis," Pattern Recognition, vol. 43, no. 1, pp. 267 – 279, 2010.

[30] N. hsien Lin, T. hao Huang, and B. yu Chen, "3d model streaming based on jpeg 2000," IEEE TCE, p. 2007.

[31] J. Peng, C.-S. Kim, and C.-C. J. Kuo, "Technologies for 3d mesh compression: A survey," Journal of Visual Communication and Image Representation, vol. 16, no. 6, pp. 688 – 733, 2005.

[32] E. Jang, S. Lee, B. Koo, D. Kim, and K. Son, "Fast 3d mesh compression using shared vertex analysis," Etri Journal - ETRI J, vol. 32, 02 2010.

[33] A. Maglo, H. Lee, G. Lavoué, C. Mouton, C. Hudelot, and F. Dupont, "Remote scientific visualization of progressive 3d meshes with x3d," in Proceedings of the 15th International Conference on Web 3D Technology, Web3D '10, (New York, NY, USA), p. 109–116, Association for Computing Machinery, 2010.

[34] K. T. Chen, Y. C. Chang, P. H. Tseng, C. Y. Huang, and C. L. Lei, "Measuring the latency of cloud gaming systems," p. 1269–1272, 2011.

[35] P. Eisert and P. Fechteler, Remote Rendering of Computer Games. 01 2007. In proceedings. International Conference on Signal Processing and Multimedia Applications (SIGMAP).

[36] M. Rivi, L. Calori, G. Muscianisi, and V. Slavnić, "In-situ visualization: State-of-the-art and some use cases," 01 2012.

[37] A. Padyana, D. Sudheer, P. K. Baruah, and A. Srinivasan, "Reducing the disk io bandwidth bottleneck through fast floating point compression using accelerators," International Journal of Advanced Computer Research (IJACR), 2014.

[38] E. W. BETHEL, HIGH PERFORMANCE VISUALIZATION. CRC Press, 2016.

[39] J. Bent, S. Faibish, J. Ahrens, G. Grider, J. Patchett, P. Tzelnic, and J. Woodring, "Jitter-free co-processing on a prototype exascale storage stack," in 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–5, April 2012.

[40] S. Shi and C.-H. Hsu, "A survey of interactive remote rendering systems," ACM Comput. Surv., vol. 47, May 2015.

[41] "Virtualgl background." https://www.virtualgl.org/About/Background. online.

[42] "Visapult-2 info pages." https://dav.lbl.gov/archive/Research/visapult2/index.html. online.

[43] L. McMillan and G. Bishop, "Plenoptic modeling: An image-based rendering system," in Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '95, (New York, NY, USA), p. 39–46, Association for Computing Machinery, 1995.

[44] K. Moreland, D. Lepage, D. Koller, and G. Humphreys, "Remote rendering for ultrascale data," Journal of Physics: Conference Series, vol. 125, p. 012096, 08 2008.

[45] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno, "Protected interactive 3d graphics via remote rendering," in ACM SIGGRAPH 2004 Papers, SIGGRAPH '04, (New York, NY, USA), p. 695–703, Association for Computing Machinery, 2004.

[46] D. Abate, R. Ciavarella, G. Furini, G. Guarnieri, S. Migliori, and S. Pierattini, "3d modeling and remote rendering technique of a high definition cultural heritage artefact," Procedia Computer Science, vol. 3, pp. 848 – 852, 2011. World Conference on Information Technology.

[47] A. Ng, J. Lepinski, D. Wigdor, S. Sanders, and P. Dietz, "Designing for low-latency direct-touch input," in Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, UIST '12, (New York, NY, USA), p. 453–464, Association for Computing Machinery, 2012.

[48] Z. Liu and J. Stasko, "Mental models, visual reasoning and interaction in information visualization: A top-down perspective," IEEE Transactions on Visualization and Computer Graphics, vol. 16, pp. 999–1008, Nov 2010.

[49] K.-T. Chen, P. Huang, and C.-L. Lei, "How sensitive are online gamers to network quality?," Commun. ACM, vol. 49, p. 34–38, Nov. 2006.

[50] T. Beigbeder, R. Coughlan, C. Lusher, J. Plunkett, E. Agu, and M. Claypool, "The effects of loss and latency on user performance in unreal tournament 2003®," in Proceedings of 3rd ACM SIGCOMM Workshop on Network and System Support for Games, NetGames '04, (New York, NY, USA), p. 144–151, Association for Computing Machinery, 2004.

[51] M. Claypool, "The effect of latency on user performance in real-time strategy games," Computer Networks, vol. 49, no. 1, pp. 52 – 70, 2005. Networking Issue in Entertainment Computing.

[52] "Augmented/virtual reality revenue forecast revised to hit $120 billion by 2020." https://www.digi-capital.com/news/2016/01/augmentedvirtual-reality-revenue-forecast-revised-to-hit-120-billion-by-2020/. online.

[53] I. Grigorik, High-Performance Browser Networking. Beijing ; Sebastopol, CA: O'Reilly, 2013. OCLC: ocn827951729.

[54] D. P. Luebke, Level of detail for 3D graphics. Morgan Kaufmann.

[55] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in Proceedings of the 1997 Symposium on Interactive 3D Graphics, I3D '97, (New York, NY, USA), p. 7–ff., Association for Computing Machinery, 1997.

[56] H. T. Kung and J. A. Webb, "Mapping image processing operations onto a linear systolic machine," Distributed Computing, vol. 1, pp. 246–257, 1986.

[57] S. Shi, K. Nahrstedt, and R. Campbell, "A real-time remote rendering system for interactive mobile graphics," ACM Trans. Multimedia Comput. Commun. Appl., vol. 8, Oct. 2012.

[58] Y. Mori, N. Fukushima, T. Yendo, T. Fujii, and M. Tanimoto, "View generation with 3d warping using depth information for ftv," Signal Processing: Image Communication, vol. 24, no. 1, pp. 65 – 72, 2009. Special issue on advances in three-dimensional television and video.

[59] G. Bishop and W. R. Mark, "Post-rendering 3d image warping: Visibility, reconstruction, and performance for depth-image warping," 1999.

[60] N. Plath, S. Knorr, L. Goldmann, and T. Sikora, "Adaptive image warping for hole prevention in 3d view synthesis," IEEE Transactions on Image Processing, vol. 22, pp. 3420–3432, Sep. 2013.

[61] C. Zhu and S. Li, "Depth image based view synthesis: New insights and perspectives on hole generation and filling," IEEE Transactions on Broadcasting, vol. 62, pp. 82–93, March 2016.

[62] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, MobiSys '12, (New York, NY, USA), p. 155–168, Association for Computing Machinery, 2012.

[63] A. Boukerche and R. W. N. Pazzi, "Remote rendering and streaming of progressive panoramas for mobile devices," in Proceedings of the 14th ACM International Conference on Multimedia, MM '06, (New York, NY, USA), p. 691–694, Association for Computing Machinery, 2006.

[64] Z. Zhou, K. Chen, and J. Zhang, "Efficient 3-d scene prefetching from learning user access patterns," IEEE Transactions on Multimedia, vol. 17, pp. 1–1, 07 2015.

[65] "Ieee standard for distributed interactive simulation–application protocols," IEEE Std 1278.1-2012 (Revision of IEEE Std 1278.1-1995), pp. 1–747, Dec 2012.

[66] A. Chan, R. W. H. Lau, and B. Ng, "A hybrid motion prediction method for caching and prefetching in distributed virtual environments," in Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST '01, (New York, NY, USA), p. 135–142, Association for Computing Machinery, 2001.

[67] "Source multiplayer networking." https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking. online.

[68] S. Lazem, M. Elteir, A. Abdel-Hamid, and D. Gracanin, "Prediction-based prefetching for remote rendering streaming in mobile virtual environments," in 2007 IEEE International Symposium on Signal Processing and Information Technology, pp. 760–765, Dec 2007.

[69] "@NVIDIAGeForce", "G-sync monitors: Best gaming pc ever built." https://www.nvidia.com/en-gb/geforce/products/g-sync-monitors/. online.

[70] "clumsy 0.2, an utility for simulating broken network for windows vista / windows 7 and a." https://jagt.github.io/clumsy/index.html. online.

[71] M. Pietroforte, "Free: Tmnetsim network simulator - simulate network latency and packet loss," 4sysops, Jun 2016.

[72] B. F. Janzen and R. J. Teather, "Is 60 fps better than 30? the impact of frame rate and latency on moving target selection," Conference on Human Factors in Computing Systems - Proceedings, p. 1477–1482, 2014.

[73] A. Steed, "A simple method for estimating the latency of interactive, real-time graphics simulations," Proceedings of the ACM Symposium on Virtual Reality Software and Technology, VRST, p. 123–129, 2008.

[74] S. Friston and A. Steed, "Measuring latency in virtual environments," IEEE Transactions on Visualization and Computer Graphics, vol. 20, no. 4, p. 616–625, 2014.

[75] S. Friston, E. Griffith, D. Swapp, A. Marshall, and A. Steed, "Profiling distributed virtual environments by tracing causality," 2018 IEEE Conference on Virtual Reality and 3D User Interfaces (VR), 2018.

[76] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," Computational linguistics, vol. 18, no. 4, pp. 467–479, 1992.

[77] A. Vlasblom, "Text prediction using n-grams." https://rstudio-pubs-static.s3.amazonaws.com/96252_bd61a0777ad44d04b619ce95ca44219c.html. online.

[78] E. Kouloumpis, T. Wilson, and J. Moore, "Twitter sentiment analysis: The good the bad and the omg," in In The International AAAI Conference on Weblogs and Social, 2011.

[79] A. Pak and P. Paroubek, "Twitter as a corpus for sentiment analysis and opinion mining.," in LREc, vol. 10, pp. 1320–1326, 2010.

[80] E. Y. Ha, J. P. Rowe, B. W. Mott, and J. C. Lester, "Goal recognition with markov logic networks for player-adaptive games," in Seventh Artificial Intelligence and Interactive Digital Entertainment Conference, 2011.

[81] S. C. Bakkes, P. H. Spronck, and H. J. Van Den Herik, "Opponent modelling for case-based adaptive game ai," Entertainment Computing, vol. 1, no. 1, pp. 27–37, 2009.

[82] S. Rabin, AI Game Programming Wisdom 4. Course Technology, 2014.

[83] I. Millington, Artificial intelligence for games. CRC Press, 2019.

[84] A. Majumdar, "Lstm network to predict mouse movements: training, prediction and interactive dataset generation." https://github.com/abhijitmajumdar/Mouse_tracking_predictor. online.

[85] "Dictionary class (system.collections.generic)." https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?redirectedfrom=MSDN&view=netframework-4.8#Remarks. online.

[86] "Prednet." https://coxlab.github.io/prednet/. online.

[87] "Fast-paced multiplayer (part i): Client-server game architecture."

[88] A. Streicher and J. D. Smeddinck, Personalized and Adaptive Serious Games, pp. 332–377. Cham: Springer International Publishing, 2016.

[89] M. Hadlow, "Message queue shootout!," Jan 1970.

[90] A. Patney, J. Kim, M. Salvi, A. Kaplanyan, C. Wyman, N. Benty, A. Lefohn, and D. Luebke, "Perceptually-based foveated virtual reality," in ACM SIGGRAPH 2016 Emerging Technologies, SIGGRAPH '16, (New York, NY, USA), Association for Computing Machinery, 2016.