Abstract

Networking has become a key technology within digital games, as multiplayer interaction has become an almost ubiquitous feature. The time required to transmit data over a network can lead to usability problems in networked games, such as inconsistency between players’ views of a virtual world and potentially jarring corrections when inconsistencies are detected. Implementing a good consistency maintenance scheme is therefore critical to game play. Sadly, problems with consistency remain a regular occurrence in multiplayer games, and these problems can have a huge impact on player experience.

This research provides guidance for implementing consistency maintenance through a series of user studies and through the establishment of a design space for consistency maintenance. The design space takes a multi-dimensional approach to consistency maintenance and, in addition to the conventional techniques used for reducing inconsistency, includes two additional design concepts – techniques for resolving game-critical decisions in the presence of inconsistency, and techniques for repairing inconsistencies when they occur. We validate the design space through a series of user studies demonstrating the trade-offs among the various techniques and confirming that it is not sufficient to focus on consistency alone. We also develop a set of metrics that provide an empirical comparison among the different techniques for consistency maintenance. We show that consistency algorithms that provide the best consistency do not necessarily lead to the best user experience. Players prefer smooth animation and prefer the results of game-critical decisions to be consistent with their view of the game, even when this results in lower overall consistency. We also show that the three most important factors affecting the detection of corrections are the player’s locus of attention, the smoothness of the correction and the duration of the correction.
I would like to thank my supervisor Dr. Nicholas Graham for his years of support and guidance. I am privileged to have worked with him during the course of this research. He has been an excellent mentor and a source of inspiration. Thank you for your optimism, patience, and friendship.

Thank you to the people who provided valuable insight on my research. This includes my committee members Dr. Patrick Martin and Dr. Juergen Dingel as well as the expertise of Dr. Carl Gutwin.

Many thanks to all the members of the EQUIS lab: Chris, Tad, Jason, Joey, Ameer, Hamilton, Zi, Banani, Max, Christophe, Murnal, Chad, Shelly, Adrian and all the summer and exchange students. It has been a pleasure to work with you over the years. I would also like to thank everyone from the School of Computing.

Thank you to my friends and family for your encouragement. My time at Queen’s has been a wonderful experience because of all the amazing people I have met.
Statement of Originality

I hereby certify that all of the work described within this thesis is the original work of the author. Any published (or unpublished) ideas and/or techniques from the work of others are fully acknowledged in accordance with the standard referencing practices.

Cheryl Savery

September 2014
# Contents

## Chapter 1 Introduction

1.1 Problem ......................................................... 3
1.2 Research Approach ........................................... 5
1.3 Steps in Research ............................................. 5
1.4 Evaluation ...................................................... 6
  1.4.1 Metrics for Comparing Techniques for Reducing Inconsistency 7
  1.4.2 Consistency Maintenance Design Space .......................... 7
  1.4.3 Guidelines for Repairing Inconsistencies ....................... 8
  1.4.4 Networking Toolkit ........................................ 9
1.5 Contributions to Knowledge ................................... 9
1.6 Thesis Outline ............................................... 10

## Chapter 2 Background and Related Research

2.1 The Consistency Maintenance Problem in Games ............... 13
  2.1.1 Consistency Maintenance Requirements ..................... 13
  2.1.2 Bandwidth Constraints ................................... 15
  2.1.3 Latency ................................................... 15
  2.1.4 Cheating .................................................. 16
  2.1.5 Game Architecture ....................................... 17
2.2 Related Problems in Other Domains .......................... 19
  2.2.1 Update Consistency ...................................... 20
  2.2.2 Replica Consistency ...................................... 22
  2.2.3 Groupware ................................................. 22
2.3 Consistency Maintenance Techniques for Networked Games .... 23
2.3.1 Predictive Techniques .............................................. 24
2.3.2 Delayed Input Techniques ....................................... 27
2.3.3 Time Offsetting Techniques .................................... 30
2.3.4 Techniques for Absolute Consistency ......................... 32
2.3.5 Summary of Techniques .......................................... 34
2.4 Game Networking Toolkits .......................................... 36
2.5 Player Performance and Experience ............................... 37
  2.5.1 Latency .......................................................... 37
  2.5.2 Player Experience ................................................ 39
2.6 Summary ............................................................... 40
Chapter 3  Consistency Maintenance Design Space ............... 42
  3.1 Reducing Inconsistency .............................................. 44
  3.2 Decision-Making ..................................................... 45
  3.3 Error Repair .......................................................... 48
  3.4 Design Consequences ............................................... 49
Chapter 4  Comparing Techniques for Reducing Inconsistency ... 51
  4.1 Types of Consistency Requirements ............................... 52
  4.2 Consistency Maintenance Metrics ................................. 53
    4.2.1 State divergence .............................................. 54
    4.2.2 Propagation delay ............................................. 55
    4.2.3 Corrections ..................................................... 55
    4.2.4 Response Time ................................................ 56
    4.2.5 Animation Delay ............................................... 57
    4.2.6 Summary ........................................................ 58
  4.3 Consistency Maintenance Algorithms ............................. 58
    4.3.1 Centralized Serialization .................................... 59
    4.3.2 Prediction (Dead Reckoning) ................................. 60
    4.3.3 Delayed Input (Local Lag) ................................. 60
    4.3.4 Time Offsetting (Remote Lag) .............................. 60
Chapter 4  Simulation Experiment .......................... 61
  4.4.1 Experimental Design and Methods ....................... 62
  4.4.2 Snagger Game ........................................... 62
  4.4.3 Simulated Network Latency ............................... 63
  4.4.4 The Algorithms ........................................ 64
  4.4.5 Metrics ................................................ 64
  4.4.6 Experimental Procedure ................................ 65
  4.5 Results ................................................. 65
  4.6 Observations and Analysis ................................ 67

Chapter 5  User Study: Consistency Maintenance Design Space 69
  5.1 Experiment .............................................. 70
    5.1.1 Game Scenarios ...................................... 70
    5.1.2 Participants ......................................... 72
    5.1.3 Procedure ........................................... 72
    5.1.4 Setup and Apparatus ................................ 74
    5.1.5 Study Design ........................................ 74
  5.2 Results ................................................ 75
    5.2.1 Q1: Does better overall consistency in a networked game always lead to better experience? ......................... 75
    5.2.2 Q2: How does the decision perspective (client or server) affect player perception of critical game events? ....................... 78
    5.2.3 Q3: How do different error repair strategies (warping or smooth correction) change experience? ......................... 81
  5.3 Summary of Results ...................................... 84
  5.4 Discussion .............................................. 85

Chapter 6  User Study: Repairing Inconsistencies 88
  6.1 Background .............................................. 89
  6.2 Error Repair Study ...................................... 91
    6.2.1 Q1: How Noticeable and Annoying are Corrections? .......... 92
List of Tables

4.1 Consistency maintenance metrics results summary ............................................. 66
5.1 Player experience questions for Paddle Blasters .............................................. 77
5.2 Player experience questions for Eliminate ....................................................... 80
5.3 Player experience questions for Speed Daemons .............................................. 81
6.1 Summary of game conditions ........................................................................... 92
6.2 Summary of participant responses .................................................................... 100
# List of Figures

1.1 Player views in World of Warcraft .................................................. 4

2.1 Typical client-server architecture ..................................................... 17
2.2 Peer-to-peer architectures ............................................................... 18
2.3 Dead reckoning ............................................................................. 24
2.4 Client-side prediction ................................................................. 27
2.5 Local lag ..................................................................................... 28
2.6 Bucket synchronization ............................................................... 29
2.7 Remote lag .................................................................................. 30
2.8 Local perception filters ............................................................... 32
2.9 Tradeoff between responsiveness and consistency ......................... 35
2.10 Taxonomy of player actions ......................................................... 37

3.1 Consistency maintenance design space ............................................. 43
3.2 Corrections to incorrect states ....................................................... 47

4.1 Consistency algorithms ................................................................. 59
4.2 Snagger two player game .............................................................. 61

5.1 Paddle Blasters: Cooperative canoeing game ............................... 70
5.2 Eliminate: First-person shooter game ............................................. 71
5.3 Speed Daemons: Racing game ....................................................... 72
5.4 Paddle Blasters: Performance ....................................................... 76
5.5 Paddle Blasters: Questionnaire responses overall ........................ 76
5.6 Paddle Blasters: Questionnaire responses by expertise ................... 76
Chapter 1

Introduction

Networked games offer an unprecedented level of interaction at a distance. Games permit people to closely coordinate the activities of small groups, to engage in large combats involving hundreds of people, and to view and react to the actions of opponents in real-time. An indication of just how important networking has become to the game industry is provided by Frank Gibeau, Executive Vice President, EA Mobile, who stated in 2012 that all future Electronic Arts games will have an online component [46].

In gaming, consistency maintenance refers to the techniques used to ensure that different players see the same, or approximately the same, shared state. However, despite the importance of networking in games, the programming of consistency maintenance algorithms remains a difficult problem. Network latency makes it impossible for players to experience fast response times while maintaining a completely consistent game state. Players expect games to respond in real-time and they have little tolerance for any indication of lag, such as jerky player movement, sluggish controls or strange phenomena like bullets not hitting their targets [9] or objects jumping across space [69].

In order to mask network latency, most networked games use consistency algorithms following one of three basic approaches. Some games use prediction to try to overcome latency, some use local lag to slow down input to match the latency in the network, and others use time offsets to display delayed versions of the game objects controlled by other players. These techniques are each described in detail in Chapter 2. They all work well in many situations—shown both by research studies (e.g., [1, 75, 94]) and by their adoption by virtually all commercial networked games. However, these
Chapter 1. Introduction

algorithms are difficult to code and their impact on player performance and experience is poorly understood. There is also little understanding of how players perceive the differences between the algorithms.

The research presented in this thesis provides guidance for implementing consistency maintenance through the development of a set of metrics for consistency maintenance and through the definition of an expanded consistency design space. The metrics provide an empirical comparison of the different approaches for reducing inconsistencies, while the design space takes a more holistic view of the consistency maintenance problem. Instead of just focusing on techniques for reducing inconsistencies, the design space includes two additional concepts—techniques for resolving game-critical decisions in the presence of inconsistency, and techniques for repairing inconsistencies when they occur. To justify the choice of axes in the design space, we have conducted a user study that investigated the effects of different consistency maintenance approaches on player experience and performance. This study demonstrated that players value intuitive local behaviour more than global consistency. However, it also pointed to the need for further investigation into techniques for repairing inconsistencies, particularly when large corrections are required. We conducted a second user study that focused on the problem of repairing inconsistencies and we have evaluated correction techniques over a range of conditions.

To complete this experimentation with different consistency maintenance techniques, we have developed a set of programming tools for game networking. These tools raise the level of expression for implementing consistency maintenance and simplify the implementation of game networking. The tools also allow for easy experimentation with different consistency maintenance techniques and evaluation of the trade-offs among them. The simplicity and utility of these programming tools is demonstrated through a series of code examples and through an exploration of the systems that have been developed using them.
Chapter 1. Introduction

1.1 Problem

The problem addressed in this research is that: *Existing consistency maintenance algorithms used in networked games are poorly understood, in terms of their implementation and in terms of their impact on player experience and performance.*

In recent years, networking has become a key technology within digital games, with many featuring multiplayer modes and online components. The time required to transmit data over a network can lead to usability problems such as inconsistency between players' views of a virtual world, and potentially jarring corrections when inconsistencies are detected. Implementing a good consistency maintenance scheme is therefore critical to game play.

Sadly, problems with consistency remain a regular occurrence in multiplayer games, including some top-selling titles, and these problems can have a huge impact on player experience. For example, when Battlefield 4 [6] was released in November 2013, players protested about the numerous lag issues with the game. Sites such as IGN [59] and Reddit [79] were inundated with negative comments from users. Even today, the game continues to suffer from issues such as rubber-banding, where characters are pulled back to previous positions, and players being killed despite being under cover. While Battlefield 4 is one of the most egregious and recent cases, it is certainly not unique. In a 2009 survey of 389 reviews of networked games, Pinelle et al. found that 49 of the reviews specifically cited gameplay problems related to consistency maintenance issues [76]. For example, in his review of Rainbow Six: Lockdown, William Harms commented that “there were constant issues with ghosting, where an enemy would simply vanish right before you doubletapped him” [53].

What is it about game networking that makes it so difficult to provide a consistent view of the game world to all players? The primary culprits are network latency, that is, the time required for updates to travel over the network, and limited bandwidth, meaning there is sometimes more data to be updated than can be transmitted without delays. To overcome these restrictions, games rely on a variety of techniques to maintain consistency. In some games, such as Age of Empires [10], game developers sacrifice responsiveness in order to provide a completely consistent view of the game world to all players. In other games, such as World of Warcraft [101], global consistency is sacrificed in order to provide a more responsive experience. For example, consider the views of
Chapter 1. Introduction

Figure 1.1: Two players’ views of the game world in World of Warcraft. Each player sees her own avatar running in front of the other player’s.

two players playing World of Warcraft as shown in Figure 1.1. On the left we see the view of the player controlling the purple female avatar, and on the right, we see the same scene at the same moment in time as viewed by the player controlling the male avatar. Each player sees her own avatar running ahead of her opponent’s. In this example, although the two views are different, it has little impact on the game play. However, if it was a racing game, who would be the winner, or in a shooter game, where should you aim?

Research has looked at the general effects of latency on player performance and experience in a range of networked games [8, 28, 72, 88]. These studies provide us with an understanding of the types of games that are most affected by network latency, but provide no guidance on techniques that can best mitigate the effects of latency. There are three main problems in developing a consistency maintenance scheme:

1. It is difficult for developers to select the appropriate technique for reducing inconsistency. Different techniques perform differently in different game situations; they may alter aspects of the player experience (e.g., reducing the responsiveness of local input devices). However, it is hard to fully understand the consequences of an algorithm’s trade-offs on player experience and it is hard to compare alternative algorithms.

2. Current knowledge about consistency techniques does not completely solve the problem of consistency maintenance in networked games. These techniques may introduce errors (e.g., due to inaccurate prediction) that need to be repaired, and the presence of inconsistencies
can complicate the fair and intuitive resolution of game-critical events.

3. Programming consistency maintenance algorithms necessitates the manipulation of state over time, requiring programmers to consider not just what value shared data has, but when that value was held. Current programming languages do not provide support for dealing with time, which makes implementing these algorithms complex. This creates a significant barrier to the invention, evaluation and deployment of such algorithms.

1.2 Research Approach

This thesis addresses these problems in three ways.

First, we present a set of metrics that describe the algorithmic space of techniques for reducing inconsistency. These metrics provide an objective comparison between these algorithms. This will allow game developers to better understand the trade-offs among the available techniques and to select appropriate techniques for each game situation.

Second, we expand on the algorithmic space of the above techniques by developing a design space that looks at consistency maintenance as a multi-dimensional problem. Rather than taking the usual focus on techniques for reducing inconsistency, the design space also considers how inconsistencies can be resolved when they inevitably occur.

Third, to facilitate the implementation of consistency maintenance algorithms for networked games, we provide a programming toolkit that simplifies the implementation of game networking and allows for experimentation with different consistency maintenance techniques.

1.3 Steps in Research

The focus of this work has been to develop guidelines to assist game developers in implementing consistency maintenance schemes in networked games.

The following steps have been undertaken to complete the research:

1. Develop a set of objective metrics for comparing consistency techniques in networked games.

We have defined a set of metrics that captures the important characteristics of techniques for
1.4 Evaluation

The evaluation includes the following components:

1. We have used a simulation experiment to evaluate the trade-offs between consistency techniques using the metrics described in Section 4.2. This work has been published as a full paper at GROUP’10 [85].

2. We have conducted a user study to evaluate the consistency maintenance design space described in Chapter 3. The user study validates the approach of considering the three dimensions of consistency maintenance, as opposed to the traditional approach of focusing on techniques for reducing inconsistency. This work has been published as a full paper at
Chapter 1. Introduction

CSCW’14 [86].

3. We have evaluated the trade-offs between repair techniques by conducting a user study that measured the effect of different techniques on player experience. This work has been accepted as a full paper at CHI PLAY 2014 [84].

4. We have evaluated how well the timelines model and its implementation within the Janus toolkit support implementing simple networking by presenting a range of networked games developed in the EQUIS lab and through code examples that demonstrate a variety of consistency maintenance techniques. Our work on the Janus toolkit has been published as a full paper at CSCW’11 [81] and in the Multimedia Systems Journal [83].

These components are described in more detail below.

1.4.1 Metrics for Comparing Techniques for Reducing Inconsistency

In order to compare different techniques for reducing inconsistency in terms of the user experience metrics, we have carried out a simulation experiment. The experiment consisted of replaying recorded player inputs for a simple two player game. Four consistency maintenance algorithms were tested under three different latency conditions. The five metrics described in Section 4.2 were measured for each algorithm and latency condition.

The experiment illustrated that user experience trade-offs between consistency techniques can be measured empirically through the metrics. We found that all approaches break down at a certain level of latency. All the algorithms have at least one major weakness when the round-trip latency grows above about a third of a second. We also found that the variance in latency can be worse than latency itself. As latency increases, so does the variance in message delivery times [5]. This results in negative consequences for each of the algorithms.

1.4.2 Consistency Maintenance Design Space

The consistency maintenance design space has been evaluated through the user study described in Chapter 5. The study looked at the following questions:

1. Does better overall consistency in a distributed game always lead to better experience?
Chapter 1. Introduction

2. How does the decision perspective (client or server) affect player perception of critical game events?

3. How do different error repair strategies (warping or smooth correction) change experience?

This experiment yielded a number of surprising lessons such as best performance not always leading to the best player experience with players preferring the version with the smoothest animation over the version that gave them the highest score. We also found that regular game players were much more likely to perceive anomalous behaviour due to latency than were novice players. In the study, the expert gamers rated intuitive local behaviour as more important than global consistency. This validates the approach of considering all three dimensions of consistency, as opposed to the traditional approach of focusing on global consistency alone.

1.4.3 Guidelines for Repairing Inconsistencies

In networked games, inconsistencies can be repaired immediately, possibly resulting in a confusing or jarring experience, or the correction can be made more smoothly over time. There is little guidance available as to which of these broad approaches (and their many variants) best reduces the impact on player experience. To address this, we conducted an experiment that provides answers to three broad questions:

1. How noticeable and annoying are corrections in networked games?
2. How do different gameplay conditions affect a player’s reaction to these corrections? and
3. What techniques may make the corrections less intrusive for game players?

We built a simple game in which corrections occurred at random intervals during the game. In our study, the players were asked to identify when corrections occurred and the degree to which each correction was distracting or confusing.

We found that the most significant factor affecting the detection of corrections was the player’s locus of attention. Distractions that did not shift the player’s focus had little impact on the number of corrections detected.

We found that for the corrections in our study, smooth corrections were vastly superior to warping and that it was significantly better to make gradual adjustments to the correction rate rather than performing the correction at a constant rate.
Chapter 1. Introduction

Finally, we found that the ability of players to detect corrections is based on the speed of the correction, not on the magnitude of the correction. When an entity is within the player’s locus of attention, its speed can be adjusted by approximately 50% relative to its normal speed. Significantly higher adjustments can be made when the entity is outside the locus of attention.

1.4.4 Networking Toolkit

The simplicity and the range of expression of the timelines model has been demonstrated through a series of code examples that show how it can be used to easily implement the following techniques:

- dead reckoning,
- local lag,
- remote lag,
- local perception filters,
- smooth corrections, and
- alternate frames of reference for game-critical decisions.

We show that by merely adjusting one or two parameters, a developer can easily switch from dead reckoning to local lag or remote lag. This provides some indication that the Janus toolkit allows for a level of experimentation not available in other game networking toolkits.

The simplicity of the Janus toolkit for implementing simple networking is further demonstrated through its use by over a dozen developers within the EQUIS lab to implement networking in a range of multiplayer games. This experience provides evidence that using the Janus toolkit can simplify the implementation of networking in games.

1.5 Contributions to Knowledge

This research contributes novel ideas and knowledge to the field of networked games. There are four major contributions:

1. Inconsistency Metrics: We define a set of metrics that describe the algorithmic space of techniques for reducing inconsistency and provide a means of performing an empirical comparison among the different approaches.
Chapter 1. Introduction

2. Design Space for Consistency Maintenance: We show that the characteristics of the techniques for reducing inconsistency alone are not enough to determine the effects on player experience. We establish a design space for consistency maintenance that includes two new design concepts—techniques for resolving game-critical decisions in the presence of inconsistency, and repair technique. Through a user study, we provide empirical evidence of the trade-offs embodied in this design space and test the effects of some of the choices presented in the design space.

3. Techniques for Repairing Inconsistencies: We show how different gameplay conditions affect a player’s reaction to corrections and techniques that may make the corrections less intrusive for game players.

4. Game Networking Infrastructure: The Janus toolkit provides a means for implementing basic networking in computer games. The toolkit also provides the infrastructure for experimentation with consistency maintenance techniques that can be used in networked games.

1.6 Thesis Outline

In Chapter 2, we discuss background work related to this thesis. The chapter provides an overview of the techniques used for consistency maintenance in networked games, followed by a review of existing toolkits used for game networking, and a discussion of user experience in networked games with a focus on the impact of delay.

In Chapter 3, we describe a design space that explores the consistency maintenance problem from three perspectives: techniques for reducing inconsistencies, repair techniques and game-critical decisions.

In Chapter 4, we propose a set of metrics, based on related work, for evaluating the effectiveness of consistency techniques. We also describe the simulation study that was conducted to compare techniques and discuss the results from the study.

We performed two studies to evaluate the effect of different consistency maintenance techniques on player experience and performance in networked games. The first study, presented in Chapter 5, validates the consistency maintenance design space. We describe the results from the
Chapter 1. Introduction

study and discuss the implications for game design. The second study, presented in Chapter 6, investigates the trade-offs between different techniques for repairing inconsistencies. We report on how different gameplay conditions affect a player’s reaction to corrections and techniques that may make the corrections less intrusive for game players.

In Chapter 7, we introduce the timelines model for consistency maintenance and and its implementation within the Janus toolkit. In this chapter, we present a series of code examples that illustrate the simplicity of the timelines model for implementing a range of consistency maintenance techniques. In Chapter 8, we summarize the thesis and draw conclusions based on the research work. We also present areas of future work.
Chapter 2

Background and Related Research

The goal of consistency maintenance in games is to give players a sufficiently equivalent experience despite network lag, without introducing undo slowdown of game play. What can be considered sufficiently equivalent varies depending on the type of game. For example, in World of Warcraft, minor differences in character positions are inconsequential, while in chess, it is essential that both players see all the pieces in the same positions on the board. Consistency maintenance in games contrasts with concurrency control as seen in other areas of computer science where consistency and correctness are paramount.

In this thesis, we experiment with different consistency maintenance schemes and study their effects on player performance and experience. Before delving into these studies, it is necessary to first acquire an understanding of the consistency maintenance approaches that are applicable to games and the algorithms for implementing them. This chapter provides this background, presenting an overview of the various consistency maintenance schemes used in networked games. Section 2.1 begins by defining consistency maintenance in games and looking at how the game architecture, the nature of networked games and network resource constraints impact the choices of consistency maintenance techniques applicable to networked games. Section 2.2 follows by comparing consistency maintenance in games to concurrency control as applied to other areas of computer science. The various consistency maintenance techniques used in games are then described in detail in Section 2.3, followed in Section 2.4 by a discussion of the support provided by existing game networking toolkits for implementing these techniques. In Section 2.5, we discuss how latency and the choice of
consistency maintenance technique impact player performance and experience and we conclude in Section 2.6 with a summary of the tradeoffs between different consistency maintenance techniques and a discussion of open problems.

2.1 The Consistency Maintenance Problem in Games

In this section, we give a more detailed discussion of what the consistency maintenance problem is by presenting the requirements that consistency maintenance algorithms must balance, and then discuss why consistency maintenance is a challenging problem, focusing on bandwidth limitations, network latency, cheating concerns, and game architecture. As we have stated, the consistency maintenance problem in networked games is to provide players with a sufficiently equivalent experience despite network lag, without introducing undo slowdown of game play.

Networked games can refer to a wide variety of games, ranging from turn-based games such as chess and card games like Bridge and Hearts, first-person shooter games where players observe the game world from a first-person perspective and engage in combat with other players, and massively multiplayer online role-playing games involving potentially thousands of players interacting in a fantasy world [4]. In this thesis, we are primarily concerned with avatar-based games, that is, games in which the player directly controls the position and actions of an avatar (or character) representing the player in the game. We investigate the types of consistency maintenance requirements that occur during such games.

2.1.1 Consistency Maintenance Requirements

Consistency maintenance in games can be viewed as a balancing act between the properties of consistency and responsiveness. In this section, we first define these properties, and then describe how the requirements for consistency and responsiveness vary depending upon the game situation.

**Responsiveness:** Responsiveness refers to the time from when a player performs an action until she sees a response to that action, for example, the time between pressing a key on the keyboard and seeing her avatar begin to move. For common interactive tasks such as typing or mouse input, a delay of less than 50-150 ms is required [89].
Chapter 2. Background and Related Research

**Consistency:** There are several types of consistency requirements for distributed systems [31]. *Update consistency* requires that all objects remain in a consistent state when accessed by multiple transactions, while *replica consistency* requires that if data is replicated, all the replicas are maintained in an equivalent state. While these consistency requirements are applicable to some aspects of networked games, such as a player trading an item with another player or the replication of the game state from the game server to the local clients, other aspects of games require a more relaxed definition of consistency. For example, the positions of player avatars and other objects within the game world are often in constant motion and updated positions must be displayed every frame independent of whether an update has been received from the server or not. This can lead to diverging views on different clients. In this case, the degree of consistency can be measured based on the distance between the positions of an object in two players’ views of the game world.

Within a game, different situations can have differing requirements for consistency and responsiveness. Typical game situations that require consistency maintenance include:

- **Life/death decisions:** Determining whether or not one player has killed another in a first-person shooter game requires a high degree of consistency. The result must be the same across all clients. In addition, these situations also have a low tolerance for delay, as the outcome affects future player actions. For example, a player that is killed should not be able to continue to shoot at other players [65].

- **Trades:** Many games allow players to trade items from their inventory with other players. These situations also require a high degree of consistency as it should not be possible for two players to hold the same item. However, these situations have more relaxed requirements for responsiveness (e.g., it may be acceptable for the outcome of the trade to take a few seconds to be propagated to all clients).

- **Avatar movement:** In many games, the precise position of other players’ avatars has little effect on the game play, as we saw in the World of Warcraft example depicted in Figure 1.1. In other games, avatar positions are critical for determining the outcome of game-critical decisions such as the life/death decisions in the Half-Life series of games [9]. In all cases, however, responsiveness of the local avatar to player input is critical.

- **Collisions between objects:** Many games use physics engines to simulate real-world physics and
interactions between game objects. When objects collide even small differences in position can lead to large differences in the outcome. For example, on one client, two objects may pass close by each other, while on another, they collide and bounce off in different directions. This can potentially require large corrections to restore consistency between the two views.

In networked games, the network environment and the game architecture impact the types of consistency maintenance techniques that can be used. We now discuss some of the restrictions imposed by these factors.

2.1.2 Bandwidth Constraints

In networked games, network communication typically consists of many small messages containing player commands or updates to the state of entities in the game [20, 40]. The bulk of these messages relate to player position. For example, the designers of Halo: Reach state that player positions, velocities and orientations account for 50% of network traffic [2], while the designers of the game Lineage 2 claim that player movement accounts for upwards of 70% of all network traffic in online role-playing games [62].

Limited bandwidth often restricts the frequency and/or size of messages that can be passed. As a result, games may use techniques to compress or aggregate messages [35]. If the frequency of update messages is reduced, the motion of entities in the game could appear jerky as the entities jump from the previous state to the newly updated state. To compensate for this, clients usually attempt to calculate the position of entities between updates to provide for smoother animation [57, 58]. In Section 2.3, we discuss these calculations in detail.

2.1.3 Latency

Latency refers to the time required for a message to propagate from one client to another. Latency can never be totally eliminated and thus some amount of latency will exist in any networked application. Latency arises due to the time for a message to be physically transmitted, which is limited by the speed of light and the slowdown of the electrical signal within the cable. Thus, a signal travelling across the Atlantic will have a minimum latency of 25–30 ms [91]. In addition to this are routing, queueing and packet handling delays that occur at each node along the message.
Chapter 2. Background and Related Research

path. These delays can add tens of milliseconds of delay to the total message transmission time. Finally, the local node adds further latency as it reassembles packets and processes the information.

The latency between two nodes is never constant. Latency varies as messages are routed through different nodes and packets are lost. This variability in latency is known as jitter. There is some evidence that jitter can have a larger impact on user experience than does the average latency. User testing for the Age of Empires [10] found that players preferred a slower response time that was consistent to a faster response time that was continually changing. In this case, players were better able to adapt to consistent longer response times than to response times that continually sped up and slowed down. Armitage et al. have shown that for games played on real-world public servers, the amount of jitter tends to increase in proportion to the amount of latency [5]. Their study of the effect of jitter under real-world network conditions for the first-person shooter game, Quake III, found that for this particular game by the time the jitter was high enough to affect player experience, the latency was so high that the game was already unplayable [5].

2.1.4 Cheating

Cheating is a major concern in networked games, with players going to extraordinary lengths to gain advantage over their opponents. Techniques used to cheat include packet snooping, cracking encrypted client-server protocols, adjusting the speed of the local clock and even modifying the compiled binary client files [55].

Due to the prevalence of cheating, the client cannot be allowed to make any critical game decisions. For example, if the local player fires at a remote player, neither the local nor the remote player’s client can be allowed to determine whether it was a hit or not. In the peer-to-peer game Age of Empires [10], cheating is combated by requiring that all clients run exactly the same simulation. If one client deviates from the others, it is flagged as “out of sync” and the game is stopped. This prevents players from hacking and modifying the client software or the client communication stream, but it does not prevent cheating by revealing information locally.

Developers of the Half-Life algorithm [9] cite cheating as the reason for only allowing clients to pass commands (e.g., keyboard or game controller inputs) and not state information (e.g., avatar position or health points) to the central server. In Half-Life, state information is always computed
Chapter 2. Background and Related Research

Figure 2.1: Typical client-server architecture: server maintains canonical game state; local clients use prediction and server updates to determine local game state.

by the server and then passed back to the client.

The threat of cheating also eliminates the viability of using consistency maintenance techniques like social protocols as used in some groupware systems.

2.1.5 Game Architecture

The types of consistency maintenance schemes used in networked games are restricted by the game architecture. Due to security concerns, networked games are almost exclusively built using a client-server architecture with only partial replication of data to the client (Figure 2.1). As discussed in the previous section, the prevalence of cheating necessitates that the server control all critical game decisions and that the client not be allowed to access any secret information. Player inputs are passed to the server, which determines the new game state and propagates it back to the clients.

When a client-server architecture is used, the server is responsible for the game logic and calculating the canonical game state. The client software typically implements a frame loop [48] consisting of three main steps:

- Polling for inputs
- Updating the game state
Figure 2.2: Peer-to-Peer (P2P) Architecture: Each peer communicates directly with all other peers

- Rendering the frame

The frame loop must execute as quickly as possible. For example, an execution time of no more than 17 ms is required to achieve a frame rate of 60 Hz. Thus, the frame loop cannot contain any large time-consuming computations; calculations such as complex physics or AI often need to be run in a separate asynchronous thread. As well, communication with the server typically takes milliseconds to complete and must also be performed asynchronously outside of the frame loop.

Due to network latency, the client cannot wait for the next game state to arrive from the server before providing feedback to the player. Thus, the client must calculate the local game state using client-side prediction (see Section 2.3.1). However, this state may later be overwritten by the game state calculated by the server. Consistency issues arise when the client’s locally calculated state differs from the server state.

The use of multiple servers can help reduce latency between the server and client, but introduces the complexity of maintaining a consistent state among the servers [12].

An alternative game architecture is peer-to-peer (P2P) (Figure 2.2) where each peer in the network is equivalent and is able to communicate directly with all other peers. For example, in the game Age of Empires, each client sends player commands to all other clients and they all
Chapter 2. Background and Related Research

calculate the identical game state in lock-step (see Section 2.3.2). This architecture was initially popular for networked games [91]. However it does not scale well to large numbers of players, and its use is usually limited to a small number of participants or games played over a local area network [91]. Because there is no authoritative server controlling the game state, cheating can be more difficult to control. An added complication with peer-to-peer games are firewalls and proxy servers that can make it difficult for clients to connect to each other [4]. Firewalls are designed to block unsolicited network traffic. In a client-server game, there is no problem because participants connect to a central server, and return traffic from the server back to the client is expected and passes through the firewall. However, with a peer-to-peer game, if a new player wants to join the game, the request to join is considered unsolicited, and will be blocked by the firewall, unless the firewall has been configured to accept unsolicited traffic. The introducer architecture helps with this, by having clients first connect to a central node, which facilitates the opening of ports required for peer-to-peer communication [99].

Before describing consistency maintenance in games, we briefly review how consistency issues are handled in other areas of computer science such as databases [38], distributed systems [31] and groupware applications [37].

2.2 Related Problems in Other Domains

As discussed in Section 2.1, the key forms of consistency requirements are update consistency and replica consistency. Update consistency requires that all objects remain in a consistent state when accessed by multiple transactions, while replica consistency requires that if data is replicated, all the replicas are maintained in an identical state. A weaker version of replica consistency requires only that all replicas reach identical states at quiescence. In this section, we discuss these topics as they apply to databases and distributed systems and compare them with the consistency problem in games.
Chapter 2. Background and Related Research

2.2.1 Update Consistency

In databases and distributed systems, concurrency control techniques are used to maintain update consistency. Concurrency control ensures correctness by preventing overlapping transactions from interfering with each other. A transaction is defined as a “sequence of operations that form a single step taking the system from one consistent state to another” [31]. Transactions must be completed in their entirety in order to ensure correctness. Concurrency problems arise when multiple transactions interfere with each other and cause incorrect results. To ensure correctness, transactions must obey the ACID properties (atomicity, consistency, isolation and durability) [31]. That is, a transaction is guaranteed to be consistent if it meets these criteria:

- **Atomicity** – A transaction is indivisible; if any part of a transaction fails, the entire transaction fails,
- **Consistency** – The result of a transaction takes the database from one consistent state to another consistent state,
- **Isolation** – Even though transactions can be executed concurrently, no transaction can access any intermediate results of another transaction, and
- **Durability** – After a transaction is complete, the results of that transaction will not be lost even in the event of system failure.

Concurrency control protocols used in databases and distributed systems are based on the concept of serializability. To increase concurrency, transactions are allowed to execute concurrently provided the result has the same effect as serial execution. The techniques used can be categorized at either pessimistic or optimistic. Pessimistic techniques require some form of checking before beginning a transaction, while with optimistic techniques, the checking is performed after the transaction has completed its task, but before the transaction is committed.

Pessimistic techniques include the use of either locks or timestamp ordering to ensure serial equivalence.

*Lock-based protocols* prevent multiple transactions from accessing the same data concurrently. The two-phase locking protocol divides the locking process into two phases: an expanding and a shrinking phase. The protocol requires that all locks are acquired (the expanding phase) before any
locks are released (the shrinking phase). With lock-based protocols, it is possible for two processes to become deadlocked, each waiting for a lock held by the other process. Thus, a deadlock detection scheme is also required.

Timestamp-based protocols use timestamps to prevent transactions from interfering with each other. All transactions are assigned a timestamp which may represent the actual time or merely a counter. The timestamp protocol ensures isolation by not allowing a transaction with a lower (earlier) timestamp to access data after a transaction with a higher (later) timestamp. If this does occur, the transactions are rolled back and repeated with new timestamps. Because locks are not used, deadlock is avoided. However, starvation may occur if transactions are repeatedly rolled back.

Multiversion concurrency control techniques keep old values of an item when it is updated. If a transaction is aborted, all the versions it created are deleted. Either timestamps or locking may be used with this technique. The advantages of multiversion concurrency control are that it improves concurrency and that read operations that arrive too late are never rejected as they can access previous versions.

Optimistic concurrency control techniques allow transactions to proceed until they are ready to commit. Transactions are applied to a local copy and at the end of the transaction, a validation step is performed. If there are no conflicts, the data is committed to the database. This technique is based on the assumption that transactions rarely attempt to access the same data. It performs well when there is little interference among transactions.

Transaction-based concurrency control techniques are rarely used in games except in situations where transactional atomicity is required. An example of this would be a player trade consisting of a withdrawal and deposit operation pair. In games, the primary goals of consistency maintenance are to provide all players with a believably consistent view of the game world while, at the same time, providing a responsive user interface. Concurrency control requirements are generally too strong for the gaming environment. In transaction-based systems, responsiveness is sacrificed in order to satisfy consistency requirements, while in gaming, the need for a responsive interface is deemed as important as consistency [91]. The requirement for a responsive interface, thus, often precludes the use of pessimistic techniques such as locks and timestamp ordering. Also, in many cases, only one player can control the position of an object in the game, particularly the position
Chapter 2. Background and Related Research

of one’s own avatar, removing the requirement for locking.

2.2.2 Replica Consistency

Various architectures are possible for maintaining consistency among the replicas. With passive (primary-backup) replication all client requests are directed to a single replica which performs the transactions and forwards the updated database state to the backup replicas. In active replication all replicas play an equivalent role. Requests are multicast to all replicas which process the request independently. In the read-one / write-all scheme, read operations may be performed on any replica, while writes must be performed on all replicas. This scheme is not realistic as operations cannot be completed if even one of the replicas fails. Instead of writing to all replicas, a quorum of replicas is typically required to complete a write transaction [31].

As seen in Section 2.1.5, in games, the central server maintains the canonical game state, while each client holds a partial replica of the game state. As with passive replication, all changes to the game state are determined by the central server and propagated back to the players’ clients. To improve responsiveness, the client will calculate the game state locally. However, if the local replica is not consistent with the server, it will be overwritten by the server state.

2.2.3 Groupware

Groupware consists of applications that focus on human interaction and allow people to communicate, collaborate and coordinate activities [37]. Examples of groupware applications include multiuser editors, group decision support systems and electronic meeting rooms. The consistency maintenance algorithms used in groupware applications are often more closely related to gaming than are the concurrency control algorithms found in databases and distributed systems. Groupware combines approaches from databases and distributed systems with other techniques such as operational transform and social protocols which are unique to groupware systems. The specific approaches commonly used by groupware include locking, optimistic locking and operational transform, serialization, and social protocols. Locking algorithms ensure consistency by only allowing a single process to manipulate a particular element at a given time. Optimistic approaches allow operations to be enacted locally on the assumption that conflicts between concurrent operations
Chapter 2. Background and Related Research

are rare. Conflicts may be resolved using rollback schemes such as TimeWarp [66] or operational transformation may be used to transform incoming operations so that the state converges [95]. With serialization [7, 24], messages are sent to a centralized server which broadcasts the messages to all clients in a consistent order. The final technique used by some groupware applications is to do nothing and merely rely upon people’s awareness of each others’ activities and their willingness to avoid conflicts [50].

Games are also an example of real-time groupware. However, there are significant differences in the networking requirements of games compared with traditional groupware applications such as shared editors and drawing programs. First, the volume of data to be transmitted in a game is much larger than the few keystrokes per second that might be generated in a shared editor. Also, in a shared editor, the state only changes in response to input from the users, while in a game, the game state may continue to change, even in the absence of player input. Finally, the competitive nature of games and the threat of cheating, preclude the use of social protocols to prevent conflicts.

In this section, we have seen a progression of algorithms. Concurrency control techniques sacrifice responsiveness in order to guarantee consistency. Techniques for replica consistency attempt to maintain a consistent state across multiple replicas. Finally, groupware applications improve responsiveness by relaxing the requirement for overall consistency. Groupware applications may allow small differences to persist and sometimes rely on the use of social protocols to avoid conflicts. Games build upon the algorithms used in traditional groupware by including techniques such as prediction [57, 58], local lag [64], and remote lag [9]. In the following section, we describe these algorithms in detail.

2.3 Consistency Maintenance Techniques for Networked Games

We identify four classes of algorithms used for consistency maintenance in games.

- Predictive algorithms such as Dead Reckoning [57, 58]
- Algorithms which delay the processing of local events such as Local Lag [64] and Bucket Synchronization [10]
- Algorithms which alter the time frame for remote entities such as Remote Lag [9] and Local
Figure 2.3: Using dead reckoning, the position of an entity is predicted based upon its previous position and velocity

Perception Filters [87, 91], and

- Algorithms which guarantee absolute consistency such as TimeWarp [66], Operational Transform [36, 95], Locking [31] and Serialization [31]

The key features of each of these groups of algorithms are presented below.

2.3.1 Predictive Techniques

Predictive techniques for consistency maintenance are known as dead reckoning. The term dead reckoning originates from traditional navigation where the path of an object (ship) is estimated based upon the direction and speed observed at its last known position. Dead reckoning is typically used for rendering entities controlled by a remote client or server. Due to the possibility of cheating, local clients often use another form of prediction known as client-side prediction for determining the position or state of locally controlled entities. These two techniques are discussed separately below.

Dead Reckoning

Dead reckoning is commonly used in distributed interactive simulations to reduce the number of update packets due to position messages. The IEEE distributed interactive simulation standard defines a protocol for dead reckoning [57, 58] whereby an extrapolation equation is used to estimate the position of an entity and instead of transmitting an update packet following each movement of the entity, updates are only transmitted when an error threshold is exceeded. In its simplest
form, the most recent position and velocity of an entity are used to predict subsequent positions of that entity. The local client maintains two types of information about each entity it is controlling. First, it stores the actual position or state of the entity, and second it stores sufficient information to determine the position that would be estimated by remote players. Thus, when the local client calculates a new position, it is able to compare the actual position to the remote players’ estimated position and determine if an update must be transmitted. In Figure 2.3, the actual trajectory of a game entity is shown as a curved grey line. The positional updates received by the remote client are indicated by the darker tanks. After the update is received, the remote client continues to estimate the current position of the tank assuming that it continues along at the same speed and in the same direction. These predicted positions are indicated by the lighter grey tanks. When the object is travelling at a constant velocity, the prediction is accurate. However, when the entity changes either speed or direction, errors will occur.

The dead reckoning extrapolation equation is usually either first order (based on position and velocity) or second order (based on position, velocity and change in either speed or direction) [91]. The equation may also be based upon only the most recent update, or it may be based upon two or more previous states. No one extrapolation technique is best in all situations. The choice of equation depends upon both the type of game and the type of motion [75]. The use of higher order polynomials has been found to work well for entities such as a vehicle where the acceleration or change in direction is smooth. However, in predicting the motion of a human avatar for example, higher order polynomials can result in increased errors if the player erratically changes direction [91].

Selecting an appropriate extrapolation equation and error threshold impacts the accuracy of dead reckoning techniques. Research involving dead reckoning has focused on two main areas:

- improving the accuracy of dead reckoning [1, 34], and
- using adaptive algorithms to automatically adjust the extrapolation equation and/or the error threshold [13, 90].

One technique used to improve the accuracy of dead reckoning is pre-reckoning [34]. The technique attempts to predict when the error in the position of an entity is likely to soon exceed its error threshold and send an update immediately instead of waiting until the threshold is actually exceeded. Typical game situations that would trigger an immediate update would occur when a
moving entity stops; when a stationary entity begins moving; or when an entity suddenly changes direction.

The degree of consistency among clients using dead reckoning can also be improved by including clock synchronization as part of the algorithm. Typically with dead reckoning, when a position is received from a remote client, that position is set as the position on the local client at the current local time. The dead reckoning algorithm does nothing to take into account the delays caused by network latency. By synchronizing the clocks on the local and remote clients, Aggarwal et al. [1] have shown that the network latency can then be accounted for and the accuracy of dead reckoning increased.

Adaptive techniques include the use of position based history [90] to analyze the type of motion exhibited by an entity. The extrapolation equation used then changes based on the type of motion. For example, if the entity is moving with constant acceleration, a higher order calculation would be used. If the entity suddenly switches to a more erratic form of motion then only first order terms would be used to calculate position.

Adaptive techniques have also been used to adjust the size of the error threshold used for transmitting updates. These techniques continually assess the game situation to determine when a greater or lesser degree of accuracy is required. For example, if a remote object is near the local player, then less error may be tolerable than if the object is further away and direct interaction with the object is not possible. Cai et al. [13] have proposed an auto-adaptive dead reckoning scheme where the error threshold changes depending upon the game situation and also upon factors such as network congestion and bandwidth availability.

Client-side Prediction

Due to the prevalence of cheating in networked games [55], local clients are usually not allowed to make any game critical decisions. This can include determining the true position of the local player’s avatar. However, if the local player had to wait for a response from the game server following each movement command, the lack of responsiveness would likely be considered unacceptable.

To improve responsiveness, local clients use client-side prediction [9] to predict the position of the local avatar. With client-side prediction, the client sends commands (keyboard or game controller inputs) to the server and the server calculates the new game state which is sent back
Chapter 2. Background and Related Research

Figure 2.4: Client-side prediction applies local commands (C1, C2, C3, etc.) to the last acknowledged state to calculate the current local state [9] to the client. The client maintains a list of the commands that have been sent to the server and the exact time at which each command occurred. As shown in Figure 2.4, the client begins with the last acknowledged state received from the server, applies all the commands that have occurred subsequent to that state and calculates the state to be displayed. This is repeated for each frame. When a new state update is received from the server, the commands that occurred before that state are dropped.

In most cases, the state calculated using client-side prediction will be identical to that received from the server. However, if for example, local entities can collide with remote entities, it is possible for the local client to predict a different outcome than the server. The resulting inconsistencies can lead to large corrections to the local state.

The key advantage of predictive techniques is that they improve responsiveness. However, they can lead to large inconsistencies if the prediction in wrong. These techniques work best when fast response times are required, and either the likelihood of a correct prediction is high or the consequences of an incorrect prediction are low.

2.3.2 Delayed Input Techniques

While predictive techniques can lead to inconsistencies in the game state, delayed input techniques such as local lag and bucket synchronization take the opposite approach. With these techniques,
Figure 2.5: Local Lag: A command initiated at one site is executed simultaneously at all sites

the goal is to reduce or eliminate inconsistencies by delaying local actions.

**Local Lag**

The key idea behind local lag [64] is to delay the execution of local commands long enough that the commands have time to propagate to all remote sites, and can then be executed simultaneously at all locations. Figure 2.5 illustrates local lag being used to synchronize the execution of commands at three sites. At the top of the figure, a command is initiated at Site 1 and broadcast to Site 0 and Site 2. The command is then executed simultaneously at all three sites. To implement local lag, a time stamp is associated with each command and the command is executed at the time of the time stamp. When a player initiates a command, instead of storing the current time, the time associated with the command is set to some point in the future. The command is then transmitted to all clients. Provided the message arrives at each remote client before the time specified in the time stamp, it is said to have arrived on time and can be executed simultaneously on all clients.

Ideally, one would set the size of the lag to be sufficient to ensure that all messages arrive on time. However in practice, with variable network latency and the potential for packet loss, it is impossible to set an upper bound on the length of time for a message to be transmitted. Thus, with local lag, although short term inconsistencies can be reduced, a mechanism is still required to repair inconsistencies when they occur.
Chapter 2. Background and Related Research

One approach for setting the amount of lag is to monitor the network latency and to continually adjust the lag to be slightly larger than the latency of the slowest client [22]. A degree of caution should be used with this approach; as observed by Bettner [10], a slower response time which is consistent is preferable to a faster response time that is continually changing. Players of “Age of Empires” were able to adapt to consistent response times of 500 ms, but found the game jerky and almost unplayable when the response time varied between 80 ms and 500 ms. These findings are supported by Shneiderman [89] who states that extreme variations in response time can be unsettling for users and should be avoided. Although large variations in response time are undesirable, other experiments reported by Shneiderman have shown that moderate variations in response times are acceptable and have little impact on task completion time and error rates.

Local lag eliminates many of the prediction errors caused by predictive techniques. However, it does so at the cost of reduced responsiveness. The use of local lag is most appropriate when the need for tight coordination among clients is more crucial than the need for responsiveness.

**Bucket Synchronization**

A variant on local lag is the bucket synchronization algorithm. This algorithm has been used in games such as Age of Empires [10], and MiMaze [33], and for sharing of legacy TV/arcade games [106].
Figure 2.7: Remote Lag: The Local View on the left shows the local player’s view of her own avatar’s movements. The Remote View on the right shows how the remote player views the motion of the same avatar. In this case all movements are delayed by 300 ms.

Bucket synchronization is based on executing commands in lock step on all clients. The communication timeline is divided into turns or buckets. Inputs generated in a given bucket are collected and applied at the beginning of a subsequent bucket. The duration of a bucket, depends upon the network latency. It must be long enough to ensure that commands reach all clients in time to execute in the appropriate bucket. In Figure 2.6, the bucket duration is set to 200 ms. Commands “a” and “b” were initiated in bucket number 1001 and are executed on all clients in bucket 1003. Similarly commands “c”, “d” and “e” are initiated in bucket 1002 and execute in bucket 1004.

For this technique to function successfully, it is vital that all commands have arrived by the time they are scheduled to execute. The game Age of Empires achieves this by having all clients determine the fastest sustainable rate at which they can run. The length of a turn is then determined by the slowest client, and a rate is selected that is sustainable by all clients. A process is used to adjust the turn length throughout the game. A late command will cause the game to pause and the length of each turn will be increased.

2.3.3 Time Offsetting Techniques

Time offsetting techniques attempt to provide the player with a fast response time and smooth animation of remote objects by displaying lagged versions of remote players and objects.

Remote Lag
Chapter 2. Background and Related Research

The technique used by first-person shooter games such as Half-Life [9] is known as remote lag. With this technique, local inputs are applied immediately to optimize responsiveness. However, remote inputs are delayed. That is, the local player sees his own actions in real-time, but sees a delayed version of all remote players. The degree to which the remote players are lagged can vary depending upon the network latency. Figure 2.7 shows an example of the remote and local players' views when using 300 ms of remote lag. The lag is generally set so that updated state information from remote players is received two frames in advance. Thus, in estimating the position of a remote player, the position can be interpolated between states which have already been received. Using interpolation to estimate the positions of remote players can result in much smoother movement than using extrapolation. This is especially true in first-person shooter games as players tend to move erratically to avoid being targeted, and in doing so make their positions difficult to extrapolate.

The use of remote lag would not be appropriate for games such as a racing game where the synchronization in time of all players is critical. However, it can be very effective in situations where fast response time is required, and where it is important that all players see the same thing (although displaced in time.)

Local Perception Filters

A variation on remote lag is the use of local perception filters [87, 91]. With the local perception filter technique, a distinction is made between player-controlled entities and passive entities whose behaviour is completely deterministic. As with remote lag, the position of the remote player is delayed. However, for a passive entity, the amount of lag applied to the object depends upon its position. When the entity is near the local player, it is drawn in real-time; however, as it approaches a remote player, a time delayed version of the entity is rendered.

Figure 2.8 shows an example of local perception filters applied to a simple Pong-type game. The dark player on the left sees his own actions in real-time and a time delayed version of the white player on the right. So far, this is identical to the remote lag technique discussed previously. The difference occurs with the ball whose position is completely predictable as it moves across the screen. The actual position of the ball is shown in light grey, while the position that the player sees on the display is shown in black. Thus for the dark player, the displayed position is very close to the actual position when the ball is on the left side of the display and it lags the actual position
Figure 2.8: Local Perception Filters: For each player, the ball is rendered in real-time when it is near their side of the display and is time delayed as it approaches their opponent’s side.

more and more as it moved across the screen. When the ball reaches the white player, the lag applied to the ball would be identical to the remote lag applied to the white player. Thus, the dark player sees the time frame of reference for the ball synchronized with the time of the white player at the moment when the two interact.

Another application for local perception filters is in the implementation of “bullet time” [92], where the game world slows down to allow players more time to make decisions.

2.3.4 Techniques for Absolute Consistency

In networked games, it is rare that absolute consistency is required between the game state of different players. However, there are some situations where absolute consistency is required. Examples include players trading or buying and selling game objects and the results of life/death decisions. Activities such as trading usually apply transaction based concurrency control techniques to ensure multiple players can’t acquire the same item. Determining whether or not a player has been hit is done by the game server and then propagated to the clients to ensure that the decision is consistent.
Chapter 2. Background and Related Research

across all players’ computers.

For instances such as these where absolute consistency is required, techniques such as *Time-Warp*, *operational transform*, *locking* and *serialization* may be used to ensure consistency.

**TimeWarp**

In order to prevent short term inconsistencies, the local lag and bucket synchronization algorithms rely upon all commands arriving by the time at which they are scheduled to execute. Unfortunately, due to jitter and packet loss it is impossible to place an upper bound on the length of time a command may require to propagate to all clients. Therefore, it is inevitable that some commands will arrive late. Although TimeWarp [66] is not known to be used in any commercial games, it is a technique which could be used to correct inconsistencies resulting from these late arriving commands. In fact, TimeWarp repairs these inconsistencies in a manner which has been shown to guarantee correctness.

With TimeWarp, each client maintains a list of commands ordered by the time at which they were executed and a list of states also ordered by time. When a command arrives late or out of order, the TimeWarp technique is applied as follows. First, the out of order command is inserted into the command list. Then, starting with the last state prior to the out of order command, all commands are reapplied to that state in a fast forward mode until the state reaches the current time.

**Operational Transform**

Operational transform [36, 95] is a consistency maintenance technique developed for use in group editors. As far as we know, it is not currently used in any networked games, although it could possibly form an alternate technique to ensure absolute consistency among players. With TimeWarp, commands are replayed whenever conflicting operations occur. By contrast, operational transform avoids replaying commands by determining a transforming operation.

The following example illustrates how operational transform is applied to a shared text editor. Consider a text document contains a string “abc”. This document is replicated at two collaborating sites: site A and site B. The following two operations are performed concurrently. Site A performs the operation: `Insert[0, "x"]` (to insert character “x” at position 0), and site B performs the operation `Delete[2, 1]` (to delete one character at position 2, i.e. “c”). Each operation will
be executed locally without transformation. However, the remote operations will be transformed before their execution.

At site A, following the local operation, the document has become “xabc”. Then the operation from site B (Delete[2, 1]) arrives and is transformed to Delete[3, 1]. Note: the positional parameter is incremented by one to include the impact of one character insertion at site A.

At site B, the document has become “ab” following the local operation. When the operation from site A (Insert[0, "x"] ) arrives again a transform is applied however the operation remains unchanged. In this case the prior execution of deleting the “c” has no impact on the operation received from site A. On both sites, the document has now become “xab”.

In summary, the basic idea of operation transform is to transform an operation into a new form according to the effects of previously executed concurrent operations so that the transformed operation can achieve the correct effect and ensure replicated documents are identical.

**Locking and Serialization**

Locking and serialization are techniques which are employed when absolute consistency is required [31]. Although the majority of game situations do not require absolute consistency, there are still important cases where it is required. Examples include situations where players are able to buy or trade or sell objects. It should never be possible for two players to acquire the same object or for a player to receive payment for an object and then not surrender that object. In these cases, locking techniques similar to those found in database transactions may be employed to ensure that the transaction is completed correctly.

In life/death decisions, serialization is used. The game server holds the master game state and arbitrates the outcome of interactions requiring life/death decisions.

Fortunately in game play when absolute consistency is required, the timing of the operation is often not critical and users are willing to accept that a trade or purchase may take a few seconds to complete.

**2.3.5 Summary of Techniques**

Consistency maintenance techniques in gaming require a tradeoff between fast response time and degree of consistency. As shown in Figure 2.9 predictive and time altering techniques have the
Chapter 2. Background and Related Research

Figure 2.9: The tradeoff between fast response time and degree of consistency. In general, time altering and predictive techniques have the fastest response times while delayed input and absolute consistency techniques have greatest degree of consistency, although there may be cases where this does not hold true.

In terms of where each type of technique is best suited, both predictive and time offsetting techniques are good choices when fast response times are required. Predictive techniques are better in cases where the state is highly predictable or where incorrect predictions can be easily corrected, while time offsetting works well when it is more important to see an exact replica of the remote state even if it is offset in time. Delayed input techniques can be used in situations where a higher degree of consistency is required and where a longer response time can be tolerated. Finally techniques for absolute consistency should be reserved for situations were there is no tolerance for any inconsistency.

In the next section, we describe the support provided by existing game networking toolkits for implementing consistency maintenance techniques.
The consistency maintenance techniques described in Section 2.3 require the manipulation of state over time, either by predicting future states or delaying inputs. This can make implementing some consistency maintenance techniques difficult and hinders experimentation with different techniques and/or combination of techniques. This is especially true for techniques like local perception filters which require the programmer to continually adjust the time frame associated with objects in the game.

There is a wide range of both commercially available and open source toolkits available for game networking. They all provide a variety of features useful for networked games, however, they provide only limited support for manipulating time. This contributes to the difficulty of implementing many consistency maintenance algorithms. These toolkits include *ZoidCom* [107] which has a special replicator that implements client side prediction, dead reckoning/extrapolation, interpolation, movement correction and local overrides. The *Torque Networking Library* [96] also includes mechanisms for interpolation and extrapolation of object positions. *Net-Z* [71] provides two models for shared objects: attribute propagation, which uses data extrapolation to reduce bandwidth requirements, and step-by-step synchronization. Other game networking libraries such as *ClanLib* [26], *NeL* [70], *OpenSkies* [73], *RakNet* [78], and *ReplicaNet* [80] provide basic networking services and include object replication, NAT punchthrough, message reliability and techniques for reducing bandwidth requirements. However, they provide little or no support for interpolation and extrapolation of shared data objects.

Due to the limited support provided by existing game networking toolkits for manipulating time, we have developed a networking toolkit that provides this support and allows for easy experimentation with consistency maintenance techniques. The toolkit is described in detail in Chapter 7 and was used to implement all the games developed for the user studies presented in this thesis.

In the next section, we discuss how network latency and the choice of consistency maintenance techniques affect user experience.
Chapter 2. Background and Related Research

Figure 2.10: Claypool’s taxonomy of player actions based on deadline and precision [30]. Actions in the lower left corner are most sensitive to latency while actions at the top right are least sensitive.

2.5 Player Performance and Experience

Networked games by their very nature are intended to be entertaining and provide an immersive virtual environment to their users. Factors such as a sluggish or inconsistent response to player inputs, jerky animation, and inconsistent states resulting in sudden jarring corrections can diminish the enjoyability of the game. As evidenced by the recent issues with lag in Battlefield 4 [6] described in Chapter 1, and the subsequent player reaction, players care deeply about the consistency problems in games.

2.5.1 Latency

The primary source of poor user experience in an otherwise engaging online game is network latency. Studies have been performed to evaluate the impact of latency on player performance and experience in racing games [74], first-person shooters [8, 17, 21, 32, 56, 77, 100], sports games [72], role-playing games [44] and real-time strategy games [88, 28].

These studies show that the type of game is a big determinant of how latency impacts player performance and of how much latency players are willing to tolerate. Claypool [29, 30] summarizes the results of these studies by grouping online games into three broad categories based upon their...
sensitivity to latency:

- first-person avatar games,
- third-person avatar games, and
- omnipresent games.

First-person avatar games include first-person shooters, racing games and some role-playing games. This type of game is the most sensitive to latency as the player requires immediate feedback to control his character’s actions. In third-person avatar games, the player typically controls a single avatar or character from the third-person point of view. Games of this type include role-playing games and sports games. Third-person games tend to be less sensitive to latency than first-person avatar games. The least sensitive are omnipresent games where the player observes the game from a god-like perspective. Games in this category include real-time strategy games and simulation games. In these games, strategy often plays a more important role than real-time actions and thus, they are less affected by high latency.

Claypool further breaks down player actions within these three game genres based on how these actions are impacted by latency. He proposes two factors: precision and deadline. Deadline is the length of time required to complete an action, for example, the time required to aim at a target and fire at it in a first-person shooter game, or the time required to move to a new location. Precision is the range in accuracy with which an action must be performed, for example how large if the target that must be sighted to score a hit. Claypool suggests that precision and deadline are the key factors for determining which actions are most impacted by latency. Actions requiring high precision and having a short deadline such as firing a sniper rifle in a first-person shooter game are significantly impacted by latencies as low as 100 ms. However, actions such as moving or exploring in a real-time strategy game require less precision and have much longer deadlines. Thus, these types of actions are only minorly impacted by latencies as high as 1000 or even 2000 ms. Figure 2.10 shows Claypool’s taxonomy of player actions in the three categories of games described above. Actions in the lower left corner are the most sensitive to latency while actions at the top right are the least sensitive.
Chapter 2. Background and Related Research

2.5.2 Player Experience

The studies described in the previous section provide a basic indication of the types of games and player actions that are most affected by network latency. However, they do not provide any guidance regarding how the choice of consistency maintenance technique will affect the user experience or performance. Developers have no control over the degree of latency found in the networks over which their users are attempting to play; however, developers can impact the user’s experience by carefully selecting consistency maintenance techniques which are appropriately matched to the type of game and to the game situation.

Kramer et al. [60] have proposed that game situations can be categorized based on the interaction context. The interaction context is defined based on three characteristics: the event type, the affected entities and the location. They then propose four quality metrics:

- **Interactivity lag**, the time for the result of a local action to be displayed locally;
- **Local drift**, the time between when the result of a remote action is displayed remotely and when it is displayed locally;
- **Global drift**, the time between when the result of a local action is displayed locally and when it is displayed on all remote clients; and
- **Ordering**, a discrete value indicating if or how events need to be ordered.

They propose that all game situations with the same interaction context will have similar consistency maintenance requirements which can be expressed in terms of these metrics. For example, avatar movement may have an interactivity lag requirement of less than 50 ms and have no requirements for local drift, global drift or ordering, while trading an item may have no requirement for interactivity lag, but require local and global drift of less than 200 ms and require the events to be processed in the correct order on all peers. This approach provides a good start for determining the type of consistency techniques that might be applicable in different game situations, but provides little guidance on how to assign appropriate values for the metrics for specific game situations or for relating the metrics to specific consistency maintenance techniques.

Fletcher et al. [43] suggest a set of four metrics for comparing consistency maintenance techniques:
Chapter 2. Background and Related Research

- **Fidelity**, the difference between the client view and the canonical (server) view;
- **Feedback rate**, the time for the result of a local action to be displayed locally;
- **Degree of warping**, the frequency and magnitude of corrections; and
- **Animation rate**, the frequency of positional updates.

These metrics allow the characteristics of different consistency maintenance techniques to be quantified. However, they provide no link to actual player experience.

Chen and Zarki [23] propose that an objective measure of quality of experience can be calculated based on three metrics: **responsiveness**, **precision** and **fairness**. They derive mathematical formulae for combining these metrics to compute an overall quality of experience score. Although being able to objectively calculate quality of experience in this manner would be remarkable, it remains to be proven how well this measure correlates to actual player experience.

We now conclude this chapter with a summary of the tradeoffs between the various categories of consistency maintenance techniques followed by a discussion of some of the open problems for consistency maintenance in networked games.

### 2.6 Summary

Consistency maintenance techniques used in networked games can generally be divided into four broad categories. Predictive techniques (Section 2.3.1) are an appropriate choice for games requiring fast response times and where modest state divergence and occasional corrections are considered acceptable. Delayed input techniques (Section 2.3.2) are a good choice in situations where it is important to have low state divergence and few corrections. Time offsetting techniques (Section 2.3.3) sacrifice state divergence for corrections. They are a good choice in cases where fast response time is required, and where it is important that all players see the same thing (although displaced in time.) Finally, techniques for absolute consistency (Section 2.3.4) provide the poorest response times and should be reserved for situations such as trades or life death decisions.

Further research is required to fully understand the effects of these algorithms and to quantify the trade-offs among them. There has been little research that explores how the choice of consistency maintenance technique affects player experience and the tradeoffs between the various
consistency maintenance algorithms. We are also aware of no work that investigates the effects of combining different consistency maintenance techniques on user performance or experience. For example, combining a small amount of local lag (perhaps 50 ms) with dead reckoning might improve consistency, while remaining undetectable by the players. These combinations such as this need to be evaluated in a variety of game situations to assess their impact on user experience and performance.

Research into consistency maintenance has usually only considered techniques for reducing inconsistency, such as predictive techniques like dead reckoning and delayed input techniques like local lag. We believe that there are other aspects of consistency maintenance that could have a larger impact on user experience, specifically the techniques used to repair inconsistencies and the choice of perspective used for game critical decisions. In this thesis, we propose a design space for consistency maintenance that includes these additional elements and conduct a set of user studies that ties the choice of techniques to actual player experience.

In the following chapter, we introduce a design space for evaluating consistency maintenance approaches that includes both decision-making and error repair.
Chapter 3

Consistency Maintenance Design

Space

In this chapter, we introduce a design space that characterizes the range of solutions to the consistency maintenance problem.

In Section 2.3, we introduced the different algorithms that are typically used for consistency maintenance in games. We saw that most networked games use consistency maintenance techniques that try to ensure that information at different nodes in the network (e.g., the locations and status of all the players in the game world) is in the same state or at least close to in the same state. There are several types of techniques that operate on different principles such as *dead reckoning* which tries to overcome latency by predicting future states, *local lag* which slows down input to match the latency in the network, and *time offsets* which delay remote player actions. However, the focus of these techniques on consistency alone fails to address all of the issues caused by network latency and diverging state.

Specifically, we have identified two additional factors that are important to player experience and performance, but that are not adequately considered by most consistency maintenance schemes. First, consistency maintenance techniques are never perfect, and so games need to make decisions in the presence of inconsistency. Second, when the techniques recognize an inconsistency and attempt to correct it, the error must be repaired at one of the nodes.
Chapter 3. Consistency Maintenance Design Space

Figure 3.1: The consistency maintenance design space consists of three axes capturing the issues that must be addressed in deciding on a consistency maintenance scheme. Each axis highlights some of the choices that are available, as opposed to representing a continuum of choices.

The first issue we name the decision-making problem. When the game must make a fast decision about a game-critical event (e.g., whether a bullet hits a target), it must often do so with the assumption that states at different nodes have diverged. Therefore, the game must choose one of the different states, or some combination thereof, to use as the basis for the decision. This choice could mean substantially different outcomes for game events, which could have a large effect on understandability and player experience.

The second issue we name the error repair problem. When an inconsistency is identified (e.g., when a prediction algorithm has moved an avatar to the wrong location), the error must be repaired, and since errors often involve a visible object in the game, the repair must be visually understandable. For example, when an avatar suddenly stops moving, there may be a delay before this information is received at other players’ computers and the avatar must then be moved back to the location where it actually stopped. Some techniques provide smoother corrections (e.g., interpolating between the incorrect and correct locations over a time period) but at the cost of
Chapter 3. Consistency Maintenance Design Space

reduced consistency, in that it takes longer for the two nodes to reach the same state.

Understanding the effects of the decision-making problem and the error repair problem are important in the design of a game’s overall approach to consistency maintenance, but there is little information available about how important these issues are, and how they affect player performance and experience. These factors are related and it is important to deal with them as a whole rather than looking at each factor individually.

In this chapter, we take the first steps in addressing this problem by describing a design space for consistency maintenance that includes decision-making and error repair as part of the overall solution for consistency maintenance in games. We begin in the next section, by introducing the three axes representing reducing inconsistency, decision-making, and repairing inconsistencies (Figure 3.1) and describe a range of approaches that help with each.

3.1 Reducing Inconsistency

As discussed previously in Section 2.3, the first line of defence in the consistency maintenance problem is to deploy an algorithm that prevents or reduces inconsistency in the first place. While a core set of algorithms has been proposed, it is not yet fully understood in what situations different algorithms should be applied. In general, these algorithms trade off game responsiveness, the frequency with which the algorithm causes jarring corrections to the player’s view, and the degree to which the views of different players diverge [85]. These algorithms follow three broad strategies, all of which have been used in commercial games: using prediction to anticipate remote updates; delaying input to allow simultaneous execution on all players’ computers, and offsetting time to allow state to diverge in a controlled fashion. More specifically:

- Predictive techniques, such as dead reckoning [58, 75], predict changes to remote state before those changes are propagated over the network. The predicted state can be used locally, compensating for the time to transmit the state over a network. However, if the prediction is incorrect, the local state must be updated, possibly in a jarring or confusing manner. Dead reckoning is commonly found in role-playing games such as EverQuest. Here, game clients predict the position of remote entities based on their last known position and velocity.
Chapter 3. Consistency Maintenance Design Space

- Delayed input techniques, such as bucket synchronization [10] and local lag [66], delay local actions to allow simultaneous execution by all clients. For example, if a player presses the “W” key to move forward, he will not see his avatar move until after the delay period, providing time for the input to be sent over the network to remote game clients. Delayed input algorithms improve consistency, at the cost of slower response to player actions.

- Time-offsetting techniques, such as remote lag [9], apply a delay to updates from remote clients. For example, in the Half-Life first person shooter game, remote players’ avatars are lagged by a constant 100 ms. Time-offsetting increases inconsistency as remote state updates are not applied immediately, but makes the degree of consistency predictable, something to which players may be able to adapt.

Each of these strategies embodies trade-offs. Prediction can increase consistency when the prediction is correct, at the cost of potentially jarring corrections when the prediction is wrong. Delayed input increases consistency at the cost of reduced responsiveness. Time-offsetting lends predictability to inconsistency, but increases inconsistency overall. Because time-offsetting displays an exact replica of the remote state which has been simply offset in time, its use can reduce or even eliminate the need for error repairs. A growing body of research attempts to determine the conditions under which the various techniques are applicable [18, 74, 94]. For example, local lag is effective up to about 100 ms of lag [94, 18]; bucket synchronization is most effective in real-time strategy games where users can tolerate small delays [30]. However, little work has been done on the criteria for selecting one algorithm over another.

3.2 Decision-Making

The second axis in the design space focuses on determining the outcome of game-critical events. Game-critical events are highly visible and have significant impact on the game’s progression. Examples include picking up a piece of treasure, crashing into an opponent’s car and blowing up, or defeating a “boss” enemy to complete a level. Game-critical events are different from, for example, moving an avatar, which can generally be easily undone or corrected, and where small differences in positions may be unnoticeable.
Chapter 3. Consistency Maintenance Design Space

The resolution of game-critical events in the presence of inconsistency can have an enormous impact on player experience and performance. In a shooter game, one player might be frustrated to miss an enemy who was clearly in his cross-hairs, while another might be upset at being hit while hiding behind a rock. When inconsistency causes players to see the world differently, the outcome of such game critical events can have unintuitive and negative consequences.

An important question in resolving game-critical events is the choice of perspective to use. If game clients all have different state, which state should be used to resolve the outcome? Game-critical decisions are usually made by a central server based on its determination of the canonical game state. A few games, however, make these game-critical decisions from the viewpoint of one of the clients. The decision is still made by the central server, but it is done based on what one of the clients sees. One important case involves actions which have a source and a target: e.g., one player shoots another, or throws a pass to another. The game-critical questions to be resolved are whether the target player was shot, or whether the target player caught the ball. Three choices of perspective are available:

- **Server perspective.** The game server maintains a canonical state. The event is resolved relative to this state, even if it disagrees with what the players see.

- **Source client perspective:** The decision is based on what the initiator of the action sees. For example, Counter Strike uses this approach for shooting, in the belief that it is more noticeable to the shooter if he erroneously misses, than to the target if he is erroneously hit [9].

- **Target client perspective resolves events based on what the target of the action sees.** This is used, for example, in Halo: Reach for determining damage when a player has activated Armor Lock [2], an ability providing the player with a few seconds of invincibility. During this period, a player would be frustrated if a grenade thrown toward her inflicted damage, and thus damage is determined from the viewpoint of the player using Armor Lock.

Note that the choice of perspective does not necessarily imply the node within a distributed system where the decision is made. For example, Counter Strike uses the shooter perspective, but the hit determination is actually made on the server (and therefore requires the server to be able to reconstruct what the shooter saw when firing) [9].
Figure 3.2: Corrections to incorrect states predicted by dead reckoning may be corrected by a) immediately warping to the correct position, b) following a straight path to a future point predicted by dead reckoning, or c) following a curved path

The perspective used for resolving hit decisions can have an enormous impact on player experience. Dick et al. have shown that in Unreal Tournament 2004, network latency above 100ms has a significant impact on game score, while in Counter Strike, player skill was the primary determinant of game score [32]. A primary difference between these games is that Unreal Tournament 2004 uses server-perspective for resolving hit decisions, while Counter Strike uses the perspective of the shooter.

When choosing a strategy for decision-making, game developers should consider which player is more likely to notice an unintuitive result, and which player is more negatively impacted by an erroneous determination.
3.3 Error Repair

The final axis of the design space deals with how to repair inconsistencies when they are discovered. Three basic approaches are used to resolve inconsistencies:

- **Correct immediately:** When predictive techniques like dead reckoning are used, the predicted state can often differ from the correct state. When a new state arrives that is significantly different from the current predicted state, the simplest solution is to immediately move, or warp, the entity to the new location as shown in Figure 3.2a. This may result in jarring visual artifacts, such as remote players “warping” to a new location. This warping can be confusing, and can distract the player’s attention to the location of the abrupt change.

- **Smoothly correct:** Here, the incorrect local state is progressively updated to the correct state. For example, a remote avatar might quickly run to the correct location rather than simply warping. Smooth correction techniques repair the inconsistency gradually over time which may appear more realistic [91] . The basic idea is to select a time in the future at which the path of the entity will coincide with the new predicted path of the entity. After selecting this target location, a possible technique would be to have the entity begin moving in a straight line toward that new location (Figure 3.2b). Although this is an improvement over immediately warping to the new location, it could potentially also have jarring effects as the entity sharply changes direction. It is possible that an even better correction might be accomplished by following a curved path to the new location, as shown in Figure 3.2c. Smooth corrections remove the effect of jarring corrections at the cost of prolonging the period of inconsistency.

- **Tolerate:** If the inconsistency doesn’t matter, it may be preferable to leave it as-is. For example, the positions of players in Blizzard’s World of Warcraft are often inconsistent, with little effect on gameplay.

There are significant trade-offs between these solutions, and to-date, these have received little attention. Correcting immediately may affect players’ immersion in the game world. A player’s attention will be drawn toward any sudden discontinuities in the position or motion of objects in the game [69]. A player may also lose context, for example, if an avatar that was in front of him is suddenly behind him and now out of his field of view.
Chapter 3. Consistency Maintenance Design Space

Smooth corrections may reduce the jarring effect of corrections [91]; however, they prolong the inconsistency, which may be unacceptable in some game situations. Fiedler suggests the rule of thumb of moving 10% toward the true position during each frame, but if the correction is large, simply warping to the new position [42]. More sophisticated forms of smooth corrections are possible; e.g. varying priorities may be given to different objects in the game to ensure the objects of higher importance deviate the least from their true position [16]. To date, there has been little effort to characterize when smooth corrections are effective, and how they should be applied.

3.4 Design Consequences

Designers need to be aware of how decisions made in one axis can affect the range of choices in other axes. We now list some examples of possible interactions. Our studies, presented in Chapters 5 and 6 explicitly test the effects of these interactions on players’ experience and performance.

- The use of local lag has positive effects on the decision-making and error repair axes. Local lag reduces inconsistency, therefore reducing the magnitude of corrections and increasing the likelihood that the client and server perspectives are the same when decisions are made. Local lag may be a good choice in situations where corrections are particularly bad, e.g., to avoid race conditions when two players are concurrently looting the same item.
- Prediction can lead to large state divergences. This can increase the importance of the choice of perspective used for decision-making, and can lead to an increasing frequency and magnitude of corrections. Prediction may be most appropriate in situations where an accurate prediction algorithm is available and where corrections do not negatively impact gameplay.
- The suitability of the remote lag algorithm depends upon the nature of game-critical decisions. If one player is more impacted by the outcome of a game decision, as in our shooter example, remote lag can allow the decision to be made from that client’s perspective. However, if all players care equally about a decision, having divergent views of the game world can be confusing. For example, in a two-person racing game, if each player saw a delayed version of the other player, both players might believe that they were the first over the finish line.
Chapter 3. Consistency Maintenance Design Space

- Smooth corrections may reduce the jarring effect of warps, but do prolong the duration of the inconsistency versus immediate correction of errors when they are discovered. Smooth corrections may therefore be a poor choice when state consistency is very important.

In the next chapter, we begin our evaluation of consistency maintenance techniques by taking a deeper look into techniques for reducing inconsistencies and presenting a set of metrics that we use to quantify the trade-offs between these techniques.
Chapter 4

Comparing Techniques for Reducing Inconsistency

Consistency problems can be addressed through consistency maintenance algorithms as described in Section 2.3 and in Chapter 3. These consistency maintenance algorithms necessarily trade off qualities of the player’s experience. For example, the local lag algorithm [94] increases the degree of consistency at the cost of reduced local responsiveness. Predictive algorithms such as dead reckoning [75] improve smoothness of animation and may reduce inconsistencies, but at the cost of greater inconsistency and potentially jarring corrections when the predictive algorithm is wrong. It is important to note that to fix one set of usability problems due to poor consistency, we frequently need to make other aspects of the player’s experience worse.

In this chapter, we hone in on the axis of our design space devoted to reducing inconsistencies. We quantify the trade-offs between the different algorithms using a set of well-defined metrics. This allows us to objectively compare the algorithms, and to understand the strengths and weaknesses of each.

We begin by describing the types of consistency requirements found in networked games.
Chapter 4. Comparing Techniques for Reducing Inconsistency

4.1 Types of Consistency Requirements

Consistency between two player’s views of the game world can diverge in three main ways – in magnitude, in time, and in rate of change. Specifically, consistency requirements can be stated in terms of:

- **State divergence**, which specifies, for a given instant in time, how much two players’ views differ. For example, a situation where a group of player avatars moves from one place to another can tolerate a higher degree of state divergence (i.e., players will not be affected by positional inaccuracies as the group moves) than a first-person shooter game where players target each other with sniper rifles (and where divergence in positions can lead to incorrect kills).

- **Propagation delay**, which indicates how long it takes the game to bring all clients to a consistent state after a change on one client. Requirements for propagation delay indicate the amount of time that a situation can tolerate an inconsistency. In a first-person shooter, high propagation delay can lead to “dead men shooting” [65] due to time to deliver a death decision. For game situations such as an avatar changing socks or transmitting a chat message, players can accept longer propagation delay.

- **Timeline divergence**, which specifies the extent to which two players’ experiences of a phenomenon differ. For example, one player might view the casting of a “fireball” spell as three seconds of casting animation, followed by an explosion effect, followed by the deduction of hit points from the target. Another player on a slower connection might see a compressed timeline, where the casting animation lasts only one second. While both players see the same final effect, the timeline has diverged. If the length of the casting animation gives the player the opportunity to take counter-measures (such as trying to interrupt the caster), then such timeline divergence can have significant gameplay consequences. Another example of timeline divergence could be one player’s view of another player’s movement, where the exact path may diverge, possibly involving jarring corrections.

Some consistency maintenance algorithms may also result in “collateral damage”, where meeting a requirement on one of the three attributes listed above may degrade other aspects of player
experience. For example, the local lag algorithm [94] reduces a game’s responsiveness, while locking or serialization algorithms may reduce the smoothness of animation. Consistency requirements must therefore be considered in the context of broader player experience.

Finally, we note that there are interface and game-design strategies that can reduce these requirements, which we do not detail here. For example, one technique that reduces consistency requirements is the idea of object pointing, in which a player selects another player as the target of an action (e.g., a spell) rather than moving a cross-hair target; this selection-based targeting dramatically increases the tolerance for state divergence.

We now introduce a set of metrics that can be used to characterize the effects of specific consistency maintenance algorithms on player experience.

### 4.2 Consistency Maintenance Metrics

As stated in Section 2.1, consistency maintenance in games requires a trade-off between the properties of consistency and responsiveness. To assess how the choice of consistency maintenance algorithm can affect user experience, we have added to these properties and identified five performance metrics. The metrics build on the work by Fletcher et al. [43] and are based on two factors:

- they correspond to commonly seen measures of the performance of groupware systems and games, and
- they can be observed and computed in real games as they execute.

Underlying the choice of these metrics is the hypothesis that each value, if allowed to become large, will in some circumstances negatively affect the player’s quality of experience in the game. These metrics allow us (and others) to develop guidelines about the consequences of a particular consistency maintenance choice. The metrics are:

- **State divergence**: The degree of inconsistency between two players’ views of the game world
- **Propagation delay**: The time from when a remote player initiates an action until the local player sees the results of that action
- **Corrections**: The number and magnitude of corrections made in order to repair an inconsistent
Chapter 4. Comparing Techniques for Reducing Inconsistency

- **Response time**: The time from when a player initiates an action until he sees the results of that action.
- **Animation delay**: The time that elapses between on-screen updates of the state the objects in the game.

Consistency maintenance techniques require trade-offs among these metrics. The following discussion summarizes these trade-offs, and where possible, looks at the range of acceptable values for each metric.

### 4.2.1 State divergence

State divergence captures the degree of consistency between two players’ views of part of the game state at a given time. For example, the state divergence of two players’ views of a single avatar could be measured as the distance between the avatar’s position on each client. An appropriate distance metric is required to measure how far the states diverge; for example, Euclidean distance might be used for positional information.

The choice of consistency maintenance technique affects the magnitude of the state divergence. For example, dead reckoning can lead to state divergence when it incorrectly predicts the future state of an entity. The degree of state divergence created by dead reckoning is influenced by several factors including the speed of the entity, the frequency and magnitude of changes in directions and the general predictability of the types of behaviour exhibited by the entity. An entity attempting to dodge enemy fire is much more likely to make sudden random movements and changes in direction than a player controlling a racing car.

Time offsetting techniques such as remote lag increase state divergence by displaying lagged versions of remote players’ avatars. The suitability of remote lag depends greatly upon the game situation. In a first-person shooter game where all hits are scored based on the shooter’s view of the game world, remote lag can greatly improve the game experience. Shooters no longer need to anticipate the future movements of their enemy and are able to score by aiming directly at their target. This may lead to some inconsistencies from the target player’s point of view, but according to Bernier [9], these are minimal compared to improved play when shooting. Thus, remote lag can...
be very effective when asymmetric roles exist among players. In a racing game where a consistent
shared state is crucial for determining who is leading and who has won the race, the use of remote
lag would be unsuitable.

Local lag minimizes state divergence and thus is most suited to games where tight consistency
is required among players.

### 4.2.2 Propagation delay

Propagation delay is the time from a player performing an action to other players’ seeing the
propagation of that action. In a first-person shooter game, this could be the time between one
player pushing the “W” key to move forward, and others seeing that player’s avatar move.

Propagation delay can never be less than the latency of the network, although some techniques
can have the effect of increasing propagation delay. Local lag and remote lag are often set to be
slightly larger than the actual network latency in order to reduce the impact of jitter. Thus, when
these techniques are used, the propagation delay will usually be somewhat longer than the network
latency.

### 4.2.3 Corrections

Corrections is a measure of the number and magnitude of modifications required to repair an
incorrectly divergent state. For example, when dead reckoning is used to estimate the position of
moving objects, a correction may be required to move the object to its correct location. As with
state divergence, a distance metric is required to measure the magnitude of a correction.

The number and magnitude of the corrections depend upon a variety of factors including:
the speed of the entity, the frequency and magnitude of changes in heading, the frequency of
positional updates and of course the network latency. When techniques are used to perform smooth
corrections, the occurrence of corrections may be barely noticeable to the player [91]. However,
jarring corrections in position, as are often used in games, can greatly diminish the user experience.
In Chapter 6, we present a study that focuses on correction techniques and their effect on player
experience.
4.2.4 Response Time

Response time represents the delay from a player performing an action to his seeing the result of that action. In a first-person shooter game, this might be the time between pushing the “W” button and seeing the avatar move forward.

Techniques such as dead reckoning and remote lag provide the fastest possible response time, with player inputs being applied immediately. Local lag and the closely related bucket synchronization algorithm intentionally delay the response to player inputs in order to achieve better consistency among clients.

According to Schneiderman, in highly interactive tasks, response times become noticeable once they exceed 50–150 ms, while with less interactive tasks, delays of up to one second are tolerable [89].

Several studies have investigated the impact of slow response time due to the use of local lag on player performance. For example, a study involving a car racing game showed no effect for delays up to 50 ms [74]. Delays of 100 ms were noticeable, but were also considered to be acceptable. At 200 ms, the game was still controllable, but the animation was not considered realistic, and at 500 ms the game was essentially unplayable. Chen et al. [18] performed a study where one user threaded a ring along a rod which was being moved by another player. In this case, there was almost no impact on user performance until the lag increased above 150 ms. In both of these studies, it was noted that more experienced users were able to compensate for the lag better than less experienced users and their performance did not decline as quickly. A third study involving a game where players controlled a fire truck and hose to put out a fire showed that a lag of less than 100 ms had only a minimal impact on player performance [94]. These studies are all consistent with Schneiderman’s findings that response times for highly interactive tasks become noticeable above 50–150 ms [89].

The above studies all involved activities where the users had direct control of objects on the screen. In real-time strategy games such as The Age of Empires, the requirements for responsiveness can be quite different. In this type of game, commands typically initiate actions such as causing troops to move to a new location or engage in battle, or the building of a new structure. User testing for The Age of Empires has shown that command latencies up to 250 ms are not noticeable,
Chapter 4. Comparing Techniques for Reducing Inconsistency

and latencies from 250 to 500 ms are still playable [10].

Thus, the response time that is acceptable is highly dependent on the game situation as well as on user experience. The amount of added delay needs to be tailored to each specific situation and must strike a balance between the needs for responsiveness and for reducing short term inconsistencies. For tasks involving direct manipulation of game entities, command latencies of up to 100 ms are usually acceptable.

4.2.5 Animation Delay

Animation delay measures the delay between on-screen updates of a particular game object’s state. For example, this might represent the time between positional updates of another player’s avatar.

Due to limitations in network bandwidth, updates to the game state are typically transmitted no more frequently than every 50 ms. To provide for smooth animation, game consistency maintenance techniques all rely on some form of interpolation or extrapolation of the game state and entity positions.

Several studies give us an indication of the animation rates required so as not to have an adverse impact on the user experience. Based on analysis with the model human processor, Card et al. [15] suggest that animation rates must be in the range of 10–20 frames per second (or 50–100 ms animation delay) for animation to be smooth. Far higher animation rates are required for fast-moving images, unless additional effects such as motion blur are used.

A review of over fifty studies involving the effects of frame rate on user performance in varying tasks shows that in general, performance degrades below 15 frames per second and there is little change in performance above this threshold [19]. For placement type tasks, this threshold is slightly higher at 17.5 frames per second. Only two of the 56 studies showed any significant change in performance above 15 frames per second.

Work by Claypool and Claypool [27] on player performance in first-person shooter games indicates that even faster animation rates are required for some game situations. As in the above studies, they found that for running and jumping actions, the frame rate had no effect when it increased above 15 frames per second. However, for shooting actions, there was a significant increase in player performance when the frame rate rose from 30 to 60 frames per second. Interestingly
though, despite the difference in player performance between 30 and 60 frames per second, the participants in the study reported little perceived difference in either quality or playability between these two frame rates. It is possible that the shooting task may be more sensitive to changes in frame rate because the scene is observed from a first-person perspective and the entire scene moves as the player adjusts her aim.

4.2.6 Summary

We believe that five metrics: state divergence, propagation delay, corrections, response time and animation delay can be used to quantify the trade-offs between the different consistency maintenance algorithms. These measures are not independent of one another. For example, high animation delay increases response time [98], poor feedthrough time (propagation delay) may lead to increased state divergence, and high state divergence may in turn lead to more severe corrections. Ideally, all of the metrics described above should have small values. However, the question of how small they have to be often depends on the requirements of the particular interactive scenario.

We now show how these metrics can be used to compare four specific consistency maintenance algorithms.

4.3 Consistency Maintenance Algorithms

In Section 2.3, we saw that there are three basic approaches for consistency maintenance: prediction, delayed input and time offsetting. We have implemented three consistency maintenance algorithms based on these approaches, as well as a fourth algorithm based on centralized serialization. The fourth algorithm was added to provide a baseline for comparison purposes. We used these four algorithms in the study described in Section 4.4. Figure 4.1 gives an overview of the operation of each of the algorithms. At the top left of the figure, we see the motion of a player’s avatar versus time. Although the motion is continuous, the positions are only sent over the network at fixed time intervals. This is represented by the X’s in the top right. In the bottom portion of the figure, we see the motion of the avatar as it would be displayed on the local player’s computer (left) and the remote player’s computer (right). When the displayed position differs from the “true” position of
Chapter 4. Comparing Techniques for Reducing Inconsistency

![Diagram of player's movement and sent over network](image)

<table>
<thead>
<tr>
<th></th>
<th>Displayed on Local Client</th>
<th>Displayed on Remote Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized Serialization</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Prediction</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Delayed Input</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
<tr>
<td>Time Offsetting</td>
<td><img src="image" alt="Diagram" /></td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Figure 4.1: The representation of the four algorithms of a given positional trace on the player’s computer and on a remote player’s computer. The assumptions are that the player position is a 1-D value changing over time, latency is a constant 50 ms, the delayed input algorithm adds 100 ms of delay, and the time offsetting algorithm adds 150 ms of delay on the remote client.

the avatar, the “true” position is also shown as a dashed line. The key properties of each of the four algorithms are described below.

### 4.3.1 Centralized Serialization

As a baseline for comparison, we implemented an algorithm based on centralized serialization. This approach is rarely used in production games, but we included it to provide a base case where clients would see the same state, although possibly offset in time due to variations in network
Chapter 4. Comparing Techniques for Reducing Inconsistency

latency. In this algorithm, movement updates are sent to the server, where they are validated and then broadcast to all clients. On each client, the player’s position is reported as the last positional update received from the server. There is no interpolation or extrapolation of the avatar positions between updates which causes the avatar position to follow a step function as shown in Figure 4.1. This algorithm also exhibits slow response times because all player inputs are processed by the central server.

4.3.2 Prediction (Dead Reckoning)

In this algorithm, updates to the player’s own position are optimistically committed locally, allowing smooth and timely local feedback (see Figure 4.1); these optimistic updates can be overruled through post-hoc server validation. On the remote client, position updates are received from the server, and therefore appear later than on the local client (i.e., increasing state divergence). To avoid the step-function effect of centralized serialization, extrapolation is used to predict the position of remote avatars by assuming that they will continue to move in their previous heading and velocity. Figure 4.1 shows that although dead reckoning provides smoother animation updates, it can lead to jarring corrections when an avatar abruptly changes direction (i.e., the spikes in the figure).

4.3.3 Delayed Input (Local Lag)

The local lag algorithm addresses the large state divergence and corrections of the prediction algorithm. Local input is delayed so that, despite delay from network latency, the local and remote clients see the same action at the same time. In addition, the remote client uses prediction to smooth avatar movement between updates. Figure 4.1 shows how local and remote movement are synchronized, lagging behind actual movement. Due to variance in packet delivery times, messages will sometimes fail to arrive within the bound of message delivery times. In such cases, error repairs can be applied.

4.3.4 Time Offsetting (Remote Lag)

The final algorithm is remote lag. Here, local inputs are applied immediately to optimize responsiveness. Remote inputs are delayed long enough for two positional updates to arrive. Prediction
is not required, since the algorithm interpolates between the two pending updates. Players thus view remote avatar movement exactly as it occurred (within the limits of interpolation), but offset in time.

These four algorithms, span the range of choices for reducing inconsistency within the consistency maintenance design space, including prediction, injection of local and remote delay, and a centralized serialization approach. Quantifying the trade-offs between these algorithms is the first step in evaluation of this key dimension of the design space. We now turn to the study that we carried out to test these four consistency maintenance algorithms in an actual game situation, and how they affected our five metrics of user experience.

4.4 Simulation Experiment

In order to test these different consistency maintenance approaches in terms of the user experience metrics defined in Section 4.2, we carried out a simulation experiment. We evaluated these algorithms using the metrics in the context of player movement. In this scenario, the consistency
maintenance problem is that of ensuring that the positions of different players’ avatars are consistent on all clients. This is a central problem, since player position is crucial to navigation, aiming, and coordination of player actions. Player movement also accounts for a significant portion of the shared data in many games. For example, in Lineage 2, player movement accounts for upwards of 70% of all network traffic in the game [62].

The experiment helps illustrate that user experience trade-offs between consistency maintenance algorithms can be measured empirically through our metrics, and it serves as a first step towards our goal of cataloguing such trade-offs to help game designers. As we shall see, our experiment confirmed expected properties of the algorithms, as well as revealing some surprising properties.

4.4.1 Experimental Design and Methods

The experiment consisted of ten simulation trials of each of the four algorithms under three different latency conditions. The algorithms introduced above were fully implemented, and were tested using recorded game traces and recorded latency values as expanded below. The five metrics of Section 4.2 were measured under each condition.

4.4.2 Snagger Game

To collect game data for the simulations, we used Snagger, a simple networked game shown in Figure 4.2. Snagger is an example of a game requiring close coordination between two players. The game consists of two clients and a server which passes messages between the two clients. Each player has a spaceship avatar, which is moved forward using the up arrow key and rotated using the left and right arrow keys. The objective of the game is for the players to move their ships around the screen and capture aliens in a net that is strung between the two ships. To capture an alien, the players must manoeuvre their ships so that an alien is inside the loop formed by the net and then press the space bar. Snagger was developed using C#, the Janus toolkit (see Chapter 7) and Microsoft’s XNA Game Studio.

To provide consistency between trials with the four algorithms, we instrumented Snagger to run using user input read from a data file. The data file contained a sequence of keyboard states, which Snagger read as if the inputs were coming from a real keyboard. Prior to the experiment, input
Chapter 4. Comparing Techniques for Reducing Inconsistency

data was collected from live play of the game (by players unassociated with this research), over a
local area network. During the game play, their inputs were captured by recording which keys were
being pressed each time the input was polled. Ten sets of data were collected, each consisting of
1,500 inputs. Each data set corresponded to approximately 25 seconds of play.

4.4.3 Simulated Network Latency

The experiment simulated three network conditions: zero latency, low latency and high latency.
The latency conditions attempted to approximate the conditions one might experience when playing
a game over a real network, where zero latency is similar to LAN play, low latency is similar to
a server on one’s own continent, and high latency corresponds to a server on another continent.
(Although the same input was played back for all simulated network latencies, it is quite likely
that players would adjust their behaviour as the latency increases, and that the inputs may not
correspond exactly to what one would experience in a high-delay situation).

The protocol used by most network games (including Age of Empires, EverQuest and Half-Life)
is UDP [67]. In the experiment, we simulated network latency and packet loss rate. CAIDA, the
Cooperative Association for Internet Data Analysis, has reported that “ping” round trip time data
matches well with the round trip time for UDP [14]. Based on this, we used “ping” round trip
time data to approximate network latency. The servers yahoo.com and yahoo.cn were selected as
representative of a server on the same continent and of a server on another continent (based on
our North American location.) “Ping” data was recorded for a three hour period for each server.
Analysis of the data for round trip times showed that it was distributed according to a Poisson
distribution with median values of 33 ms and 323 ms for yahoo.com and yahoo.cn respectively.

The “ping” file was split into 10 parts, each long enough to be usable for one simulation trial.
In each trial, the system loaded files for the game data and network latency; as each message was
sent from the client to the server and back to the other client, the message was delayed by a time
equal to the next round trip time from the file.
4.4.4 The Algorithms

The four algorithms were implemented as described in Section 4.3. In the Half-Life game, a message containing the player’s position is sent to the server every 50 ms [9]. For consistency, we selected this frequency for all the algorithms.

The delayed input and time offsetting algorithms required us to choose values for their remote and local lag parameters.

The choice of a local lag constant should strike a balance between the negative effects of the lag itself on responsiveness of the user interface versus the state divergence and corrections resulting from late messages [66]. We chose a lag constant of 100 ms, based on Stuckel and Gutwin’s findings that this is the largest tolerable lag in highly interactive tasks [94].

For the time offsetting algorithm, the remote lag value should ensure that at least one future position is available at all times, allowing the position of the remote avatar to always be interpolated. In practice, it is impossible to set an upper bound on the time it may take for a message to arrive. We therefore set the amount of remote delay to twice the median network latency (as described above), plus (to allow for occasional lost messages) twice the time between message sends (2 * 50 ms in our case). The remote player delay values used in the experiment were therefore set at 100 ms, 166 ms and 746 ms for zero, low and high latency respectively.

For the prediction and centralized serialization algorithms, no additional delay was used.

4.4.5 Metrics

The five metrics were calculated as follows:

Response time was calculated by storing the time each input was polled and then determining the first time at which that input had an impact on the displayed position of the local avatar. Our measure did not include operating system time (the time that elapsed between when a key was pressed and when the input was polled) or the time between invocation of screen update operations and the appearance of new information on the physical display.

Propagation delay was calculated by storing the time associated with each input by the remote player and then determining the first time each input had an impact on the displayed position
of the remote avatar on the local computer. As with response time, it did not include operating system time.

Corrections were calculated each time the local player received a message containing an updated position for the remote player’s avatar. The estimated position of the remote player’s avatar was calculated just before the new position message was processed. A second position was then calculated just after the new position message had been processed. The distance warped was calculated as the distance between these two positions and was expressed as a percentage of the screen size.

State divergence was calculated as the difference between the positions of the remote avatar on the local and remote client. State divergence was also expressed as a percentage of the screen size.

Animation delay is the time between changes in the position of the remote avatar. Whenever a screen update was invoked, if the position of the remote avatar had changed, then the elapsed time since the last position update was recorded.

4.4.6 Experimental Procedure

Each of the four algorithms was tested under each of the three latency conditions, for a total of 12 conditions. Each condition was tested ten times, and each condition used the same ten input game-data files and latency files, as described above.

For each trial, two instances of the Snagger client and the Snagger server were run on a single computer (2.66 GHz Intel i5 processor, 3 GB RAM). Each trial used different input and latency files. Values for the metrics over each trial were determined and their values recorded.

4.5 Results

The results from the experimental runs are summarized in Table 4.1. All the metrics were designed such that a small value is good and a larger value indicates poorer performance.

As shown in Table 4.1, centralized serialization has the highest animation delay, since it does not extrapolate or interpolate avatar positions between updates coming from the network. The prediction and time offsetting algorithms had the lowest response times, since they perform immediate
Chapter 4. Comparing Techniques for Reducing Inconsistency

Table 4.1: Results summary. Within each latency condition, all differences in the table are significant at the $\alpha = 0.05$ level, except propagation delay for the centralized serialization and prediction algorithms under the medium and high latency conditions.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Delay (Local/Remote)</th>
<th>Network Latency (ms)</th>
<th>State Divergence (%)</th>
<th>Prop. Delay (ms)</th>
<th>Correct. per sec (%)</th>
<th>Response Time (ms)</th>
<th>Anim. Delay (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialization</td>
<td>0/0</td>
<td>0</td>
<td>0.14</td>
<td>29</td>
<td>6.4</td>
<td>18</td>
<td>49</td>
</tr>
<tr>
<td>Prediction</td>
<td>0/0</td>
<td>0</td>
<td>0.08</td>
<td>29</td>
<td>0.4</td>
<td>1.5</td>
<td>17</td>
</tr>
<tr>
<td>Delayed Input</td>
<td>100/100</td>
<td>0</td>
<td>0.04</td>
<td>78</td>
<td>0.0</td>
<td>90</td>
<td>17</td>
</tr>
<tr>
<td>Time Offsetting</td>
<td>0/100</td>
<td>0</td>
<td>0.68</td>
<td>78</td>
<td>0.0</td>
<td>1.5</td>
<td>17</td>
</tr>
<tr>
<td>Serialization</td>
<td>0/0</td>
<td>33</td>
<td>0.33</td>
<td>63</td>
<td>6.5</td>
<td>58</td>
<td>50</td>
</tr>
<tr>
<td>Prediction</td>
<td>0/0</td>
<td>33</td>
<td>0.10</td>
<td>64</td>
<td>0.7</td>
<td>1.5</td>
<td>17</td>
</tr>
<tr>
<td>Delayed Input</td>
<td>100/100</td>
<td>33</td>
<td>0.06</td>
<td>80</td>
<td>0.1</td>
<td>88</td>
<td>17</td>
</tr>
<tr>
<td>Time Offsetting</td>
<td>0/166</td>
<td>33</td>
<td>1.09</td>
<td>145</td>
<td>0.0</td>
<td>1.5</td>
<td>17</td>
</tr>
<tr>
<td>Serialization</td>
<td>0/0</td>
<td>323</td>
<td>2.10</td>
<td>346</td>
<td>8.3</td>
<td>342</td>
<td>66</td>
</tr>
<tr>
<td>Prediction</td>
<td>0/0</td>
<td>323</td>
<td>0.50</td>
<td>345</td>
<td>3.8</td>
<td>1.5</td>
<td>17</td>
</tr>
<tr>
<td>Delayed Input</td>
<td>100/100</td>
<td>323</td>
<td>0.30</td>
<td>343</td>
<td>2.7</td>
<td>94</td>
<td>17</td>
</tr>
<tr>
<td>Time Offsetting</td>
<td>0/746</td>
<td>323</td>
<td>4.55</td>
<td>725</td>
<td>0.0</td>
<td>1.5</td>
<td>17</td>
</tr>
</tbody>
</table>

Optimistic updates. Delayed input has the highest response times (because of the injected local lag), and the lowest state divergence (as updates are synchronized on all clients.) Time offsetting has the lowest corrections value (as it uses interpolation rather than extrapolation); it also has the highest state divergence, due to the injection of remote lag without corresponding local lag.

The results allow us to summarize the strengths and weakness of the four algorithms as applied to the avatar movement problem:

- **Centralized serialization** has, as expected, no obvious benefits, performing poorly under all metrics. The performance of this algorithm clearly shows the hazards of applying generic consistency maintenance strategies for specific game situations.

- The **predictive** algorithm provides a balance between all of the metrics - one of the main advantages of this algorithm is that it has no significant weaknesses. It is an appropriate choice for games where modest state divergence and occasional corrections are considered acceptable. It would not be appropriate, however, for first-person shooter games where the erratic movement of players’ avatars can be difficult to predict.

- **Delayed input** is a good choice in situations where it is important to have low state divergence
Chapter 4. Comparing Techniques for Reducing Inconsistency

and few corrections, and where a commensurate penalty in response time is acceptable. For example, local lag is a good match with the indirect control of real-time strategy games (where target positions of game entities are specified via “click to move” input).

• The time offsetting algorithm sacrifices state divergence for corrections. This algorithm is a good choice in cases where fast response time is required, and where it is important that all players see the same thing (although displaced in time.)

4.6 Observations and Analysis

The experiment provides several lessons for developers of consistency maintenance algorithms.

All approaches break down at a certain level of latency. The experiment shows that all the algorithms have at least one major weakness when round-trip latency grows above about a third of a second. These results reinforce that consistency maintenance techniques do not compensate for latency, they only try to protect certain user or system requirements from latency’s effects. However, game design can make up for some of these limitations, such as by increasing the time scale of interaction, by reducing the precision of the weapons (no sniper rifles, only shotguns), or by slowing down movement.

Variance in latency also affects the performance of the algorithms. In our study, as the latency increased, so did the variance in message delivery times. This resulted in negative consequences for each of the algorithms. Under centralized serialization, all clients are sent the same updates at the same time; however, messages arrive at the clients at different times, causing significant state divergence. Variance in latency increases the difficulty of setting local and remote lag parameters (these trade-offs are discussed below.) A further effect is that animation delay under centralized serialization increases with latency. Messages are sent at a constant 50 ms rate, and so one might expect (as seen in the zero latency case) that animation delay would be close to 50 ms. However, due to variance in latency, it is possible for two position updates to arrive within the same interval between frame loop updates, causing the first update to be effectively lost. As the variance in latency increases, this occurs more frequently, leading to increased animation delay.

Propagation delay comes from more sources than network latency. When building games as
Chapter 4. Comparing Techniques for Reducing Inconsistency

distributed systems, it is easy to focus on network latency as the primary source of propagation delay. However, as seen from the results for centralized serialization and prediction in the zero latency case, other sources can result in significant delay. These include (1) message send delay: since positions are sent periodically (e.g., every 50 ms) and not as they occur, there can be a delay between a change being made and a change message being sent; (2) frame-loop delay: if the frame loop executes once per 17 ms, there is a delay of up to 17 ms from a message’s arrival to its being processed by the game; and (3) operating system delay: such as the time for messages to be processed, and time for the network reading thread to be scheduled.

*Adjusting algorithm parameters can enable helpful trade-offs.* Our experiment used particular fixed values for each algorithm’s parameters, but adjustments to these parameters can change the trade-offs inherent in each approach. For example, the local lag algorithm in our experiment added a constant 100 ms lag. However, any amount of lag can be used in this scheme, which gives the designer a trade-off between state divergence and local responsiveness. More local lag reduces divergence, and less improves responsiveness. It is therefore possible to use local lag to reduce some of the divergence, without sacrificing all of the local player’s ability to control their character. The appropriate settings for each algorithm depend (once again) on the human factors of the interaction: for example, adding 50 ms of local lag might not be noticed by players, but might bring the state divergence within the tolerance that is needed for a game situation. It is interesting to note that when interpolation is used, response time and propagation delay can actually be *less than* the injected lag. This is because the effect of a queued positional update influences interpolation.

The metrics described in this chapter provide an objective comparison between consistency maintenance techniques. However, they focus solely on the dimension of techniques for reducing inconsistency. Further testing is required to study all three dimensions of the consistency design space and to determine how the choice of technique relates to actual player experience. The study, described in the next chapter, takes a step in this direction by looking at how consistency maintenance techniques from the various axes can be combined and at how the choices impact player performance and experience.
Chapter 5

User Study: Consistency Maintenance Design Space

The study described in Chapter 4 focused on measuring the characteristics of specific techniques used to reduce inconsistency. This study expands on that work by taking a more holistic view of consistency maintenance based on the consistency maintenance design space described in Chapter 3. In addition to studying the core algorithms for reducing inconsistency, we also consider the problems of making game-critical decisions and of repairing inconsistencies. That is, in the presence of inconsistency, how can we make game-critical decisions that appear fair and intuitive for all players, and when inconsistencies do occur, how should they be repaired to have the least impact on player experience?

This study provides a better understanding of how all three dimensions of the consistency maintenance design space affect player experience and performance. In particular, we designed the study to answer four questions:

Q1. Does better overall consistency in a networked game always lead to better experience?
Q2. How does the decision perspective (client or server) affect player perception of critical game events?
Q3. How do different error repair strategies (warping or smooth correction) change experience?
Q4. Does the player’s level of experience affect the results of Q1-Q3?
We addressed these questions using three custom networked games that replicate critical aspects of real multiplayer game situations (but that were otherwise kept simple to avoid the effects of strategy).

5.1 Experiment

The experiment described in this section was conducted in conjunction with researchers at the University of Saskatchewan. The software development for the games and the analysis of the results were done at Queen’s, while the study itself was conducted in Saskatchewan.

5.1.1 Game Scenarios

The user study was conducted using three two-player games. The games were limited to two players in order to facilitate scheduling and running the experiments. We believe, however, that the effects we observed in the study would be equally applicable to two players or many players in the same situation. The three games used in the study were: Paddle Blasters, a cooperative canoeing game, Eliminate, a first-person shooter, and Speed Daemons, a racing game. Each of these games is described below.

Paddle Blasters

Paddle Blasters is a cooperative two-player game that was used to investigate whether overall consistency led to better player experience (Q1). In Paddle Blasters, players attempt to paddle a canoe down a river: the goal is to keep the canoe as close as possible to the black line that zigzags down the river (Figure 5.1). The players paddle on different sides of the canoe by pressing the
Chapter 5. User Study: Consistency Maintenance Design Space

Figure 5.2: Eliminate: FPS in which players attempt to hit their opponent while avoiding being hit themselves

space bar. If one player is paddling, the canoe turns; when both players paddle, the canoe moves ahead in its current direction.

Eliminate

Eliminate is a first-person shooter game used to investigate the effect of using different perspectives for game-critical decisions (Q2). In the game, players attempt to shoot their opponent while avoiding incoming shots. Simple block avatars are located on a platform separated from the other player by an open area that cannot be crossed (Figure 5.2). This restricts the movement of each player and forces the player to focus on the shooting task. The players begin the game behind a wall located in the centre of their platform. Players then use the WASD keys to move their avatar, and use the mouse to aim and shoot. To discourage the players from blindly firing without aiming, players must wait three seconds between firing.

Speed Daemons

Speed Daemons is a 2-D side scrolling game used to study techniques for repairing location errors: immediate warping or smooth correction (Q3). Each player controls a race car and attempts to pick up coins while avoiding mud puddles (Figure 5.3). Players earn one point for each coin they collect; colliding with puddles causes a loss of two points. The game scrolls from right to left at a constant rate with new coins and mud puddles appearing from the right.
Chapter 5. User Study: Consistency Maintenance Design Space

Figure 5.3: Speed Daemons: Racing game in which players attempt to collect coins while avoiding mud puddles

5.1.2 Participants

For the study, twenty-six participants were recruited from the University of Saskatchewan student population. Thirteen of the participants were frequent and experienced gamers, and 13 were non-gamers. We classed participants as gamers if they played fast-paced games at least once per week. We did not require expertise in any given game. Although expertise might be specific to an individual game or game genre, we hypothesized that there is significant commonality between games in, for example, the ability to rapidly manipulate game controls, or to visually process game events. In addition, because our games are designed to focus on common game elements (moving, targeting, avoiding), there is a reasonable expectation that expertise will apply across all of our scenarios. We use gaming experience as a factor in the study described below.

5.1.3 Procedure

Participants played in pairs, seated at separate computers and able to communicate through a headset. For each game scenario, the players were shown the game and allowed to practice for up to five minutes. During this learning period no latency was added to the game. Typical network latency during the practice sessions, as measured by ping time, was 1 ms or lower.

Following the training period, in order to simulate typical wide area network conditions, synthetic latency was introduced on the network. Based on work by Armitage [3] showing that players of games such as Quake 3 actively select servers where the latency is less than 150 to 180 ms, we
chose to limit the synthetic latency to 200 ms. The latency was distributed between 50 and 200 ms according to a Poisson distribution so as to approximate real-world network conditions. The participants then played several trials. Each trial was identical except that it used a different consistency maintenance technique, which instantiated the different conditions for questions 1-3 above. The order of the trials was randomized and the conditions were balanced for order. Specifics of the trial conditions for each game scenario are described below.

**Paddle Blasters (Q1).** Participants played three different conditions each of which used a different technique for reducing inconsistency. The conditions were:

- **Condition A:** 200 ms of remote lag (Lowest Consistency between the two players’ views)
- **Condition B:** 100 ms of local lag with dead-reckoning-based prediction used when the latency exceeded 100 ms (Medium Consistency)
- **Condition C:** 200 ms of local lag (Highest Consistency)

Each condition was repeated three times for one minute each. In condition B, warping was applied to repair any incorrect states that occurred when latency exceeded 100 ms. No information regarding the specific differences between conditions was provided to the participants.

**Eliminate (Q2).** There were two five-minute trials, both of which used 200 ms of remote lag for displaying the position of the remote avatar. A remote lag of 200 ms was chosen to coincide with the maximum latency on the network, thus the results would not be affected by corrections to the position of the remote player’s avatar.

The two trials differed based on the perspective used for making the game-critical decision of whether or not a player was shot. The specific conditions were:

- **Condition A:** Hits were determined based on the shooter’s perspective (Shooter)
- **Condition B:** Hits were determined based on the canonical game state as determined by the authoritative central server (Server).

Because the players played each condition for only five minutes, we provided them with information about how the hits would be determined so that they could adjust their game play for each condition.

**Speed Daemons (Q3).** Participants played three one-minute trials in which we varied the technique used for repairing inconsistencies. The conditions were:
Chapter 5. User Study: Consistency Maintenance Design Space

- **Condition A**: Dead reckoning with immediate position warping was used to fix incorrect positions (Warping)
- **Condition B**: Dead reckoning with smooth interpolation-based corrections was used to repair inconsistencies (Smooth)
- **Condition C**: Remote lag of 200 ms was applied to the remote car position, meaning that no corrections were required for the position of the remote car. (Local Only)

In condition C, because the amount of remote lag always exceeded the network latency, there were never any corrections for the remote car. Corrections could occur to the local car when it collided with the remote car. Smooth corrections were used to repair any incorrect states for the local car caused by such collisions. No information regarding the specific differences between conditions was provided to the participants.

After each of the trials in each of the games, participants answered several Likert-style questions about their experience in that particular game (Tables 5.1, 5.2, and 5.3). At the end of each game, participants were also asked to indicate which of the conditions they preferred.

### 5.1.4 Setup and Apparatus

The study used custom software written in C# using the *Janus* toolkit (see Chapter 7) and Microsoft’s XNA Game Studio, and ran on two computers connected via a dedicated network. The system maintained a frame rate of approximately 60 FPS. Each game used two clients and a server. The server arbitrated conflicts between the two clients (e.g., if both players picked up a coin at the same time in Speed Daemons).

### 5.1.5 Study Design

We analyzed each game separately, using mixed-factorial ANOVAs to test for effects on performance measures. One within-participants factor was determined for each question (i.e., Degree of Consistency for Q1, Decision Perspective for Q2, and Correction Method for Q3, and Expertise (Gamer or Non-Gamer) was used as a between-participants factor. Player experience (the questions of Tables 1, 2, and 3) was analyzed with Wilcoxon tests on medians.

We now describe the results from the study.
Chapter 5. User Study: Consistency Maintenance Design Space

5.2 Results

The results are organized as follows – for each question, we first consider the performance measures, then the player experience questions and then present a brief interpretation of the results. We conclude the section with a discussion and summary of the overall results from the study.

5.2.1 Q1: Does better overall consistency in a networked game always lead to better experience?

Player Performance

The Paddle Blasters game compared three different consistency maintenance schemes, providing different amounts of overall consistency between the two clients’ views (Degree of Consistency). As we saw in our study in Chapter 4, Remote Lag produces highest state divergence or the least overall consistency because the remote player’s actions are delayed by 200 ms. Local Lag produces the lowest state divergence or highest consistency because both players’ actions are delayed by the same amount, so both players have a (nearly) identical view of the game. The hybrid approach provides consistency between these two extremes.

We analyzed the effects of Degree of Consistency on the pair’s shared score (we used the mean of the pair’s best two scores for each technique; note that lower scores are better in this game). The mean for each condition including an indication of the standard error is shown in Figure 5.4. One-way within-subjects ANOVA showed a significant effect of Degree of Consistency (F_{2,49}=3.81, p=0.038). Follow-up pairwise t-tests indicated that scores were significantly worse in the High Consistency condition (200 ms Local Lag) compared to the Medium Consistency condition ((100 ms Local Lag + Prediction) (p=0.007) and there was a near significant difference between the High Consistency condition and the Low Consistency condition (Remote Lag) (p=0.089). The p values for all conditions showing a significant difference are indicated above the bars in Figure 5.4 (and in all other figures).

Player Experience

For the experience questions (Table 5.1), Friedman tests showed a main effect of Degree of Consistency for the ‘overall’ question ($\chi^2=6.72$, p=0.035) and near-significant effects for the ‘jerkiness’
Chapter 5. User Study: Consistency Maintenance Design Space

Figure 5.4: Paddle Blasters: Overall performance (means and standard error) by degree of consistency.

Figure 5.5: Paddle Blasters: Questionnaire measures (medians), by degree of consistency.

Figure 5.6: Paddle Blasters: Questionnaire median responses by expertise (Gamer / Non-Gamer) and degree of consistency

question ($\chi^2=5.51, p=0.063$) and the ‘responsiveness’ question ($\chi^2=5.51, p=0.063$) (Figure 5.5).

In addition, we divided the data by Expertise into Gamer and Non-Gamer groups (Figure 5.6).
Chapter 5. User Study: Consistency Maintenance Design Space

Table 5.1: Player Experience Questions for Paddle Blasters

<table>
<thead>
<tr>
<th>OtherDelay</th>
<th>The other player’s paddling seemed delayed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jerky</td>
<td>The canoe moved in an unexpected jerky manner</td>
</tr>
<tr>
<td>Coordinate</td>
<td>It was easy to coordinate my paddling with the other player.</td>
</tr>
<tr>
<td>Responsive</td>
<td>The canoe movement was responsive when I pressed the space bar</td>
</tr>
<tr>
<td>Overall</td>
<td>Overall this version was as good as the practice version</td>
</tr>
</tbody>
</table>

For Non-Gamers, Friedman tests show no effect of Degree of Consistency on any question. For Gamers, Friedman tests showed a main effect of Degree of Consistency for the ‘jerkiness’ question \(\chi^2=6.78, p=0.034\) and a near-significant effect for the ‘overall’ question \(\chi^2=5.90, p=0.052\). Follow up Wilcoxon tests showed that for the jerkiness question, the Medium Consistency condition was significantly worse than either Low or High Consistency \(p<0.05\). For the ‘overall’ question, Wilcoxon tests showed no differences.

Following the trials, the participants indicated their overall preference among the three techniques. For Non-Gamers, five chose the Low Consistency condition (Remote Lag), two chose Medium Consistency (Local Lag plus prediction), and six chose High Consistency (Local Lag). For Gamers, seven chose Low Consistency, two chose Medium Consistency, and four chose High Consistency.

**Interpretation of Results**

In Paddle Blasters, the High Consistency condition (Local Lag) resulted in the worst overall score. This indicates that overall consistency does not necessarily result in the best performance. Interestingly, however, the results were not as clear-cut for player experience, with 10 players selecting the High Consistency condition as the best overall and 12 selecting the Low Consistency condition.

Contrary to previous evidence that 100 ms is the limit for local lag in direct interaction [23], we found that fewer than half of the participants noticed 200 ms of local lag. Answers to the ‘responsiveness’ question (Table 5.1) showed that 15 of the 26 participants found the High Consistency (Local Lag) condition to be as or more responsive than the Low Consistency (Remote Lag) condition. We believe that the game type has an effect on the noticeability of local lag. In this game, the local player was not the only one controlling the canoe which may account for why most players (even the gamers) did not notice the 200 ms of local delay. However, even though the participants did not always notice the lag, it did affect their score in the game.
Chapter 5. User Study: Consistency Maintenance Design Space

The Medium Consistency condition resulted in the best overall score. However, only four of the 26 participants selected this as their preferred version of the game. This counter-intuitive result is best explained by one participant’s comment: “The jerkiness of [this condition] was off-putting, but it played the best.”

Our results show that players have definite opinions about the play experience of different consistency maintenance techniques, but that these are not strongly related to degree of consistency (players preferred either the least or the most consistent version). In addition, it is clear that the visual jerkiness of the game is an important factor (an issue we consider in more detail below).

5.2.2 Q2: How does the decision perspective (client or server) affect player perception of critical game events?

Player Performance
The Eliminate game was used to explore the importance of the perspective from which game-critical decisions were made. The two conditions were Server, where decisions are made based on the server view of the game, and Shooter, in which decisions are made based on what the shooter saw. The Server condition requires players to “lead” or predict the position of a moving enemy in order to score a hit, while in the Shooter condition, players aim directly at their opponent’s avatar. We analyzed the effects of factor Decision Perspective on game performance (individual accuracy and score). A 2x2 ANOVA showed a main effect of Decision Perspective on score ($F_{1,24} = 4.39$, $p=0.042$) and on accuracy ($F_{1,24} = 8.00$, $p=0.007$). As shown in Figure 5.7, both score and accuracy were lower when decisions were made based on the Server viewpoint.

ANOVA did not show a main effect of Expertise on either score ($F_{1,24} = 2.83$, $p=0.10$) or accuracy ($F_{1,24} = 0.282$, $p=0.60$). Differences between Gamers and Non-Gamers are also shown in Figure 5.7. There was no interaction between Decision Perspective and Expertise on score ($F_{1,24} = 0.68$, $p=0.41$) or accuracy ($F_{1,24} = 0.165$, $p=0.69$).

Player Experience
For the experience questions (Figure 5.8), Wilcoxon tests showed a main effect of Decision Perspective ($Z=-2.42$, $p=0.015$) only for the question “Aiming at the other player was easy.” Since Wilcoxon tests do not determine interactions, we divided the data by Expertise, and carried out
Chapter 5. User Study: Consistency Maintenance Design Space

Figure 5.7: Eliminate: Mean performance by expertise and decision perspective (Server / Shooter)

Figure 5.8: Eliminate: Questionnaire median responses by decision perspective (Server/Shooter)

Figure 5.9: Eliminate: Questionnaire median responses by expertise (Gamer / Non-Gamer) and by decision perspective

secondary analyses with the Gamer and Non-Gamer data (Figure 5.9). For Non-Gamers, Wilcoxon tests did not show a main effect of Decision Perspective for any of the questions. However, for the Gamers, Wilcoxon tests showed a main effect of Decision Perspective for three questions: “Aiming
Chapter 5. User Study: Consistency Maintenance Design Space

Table 5.2: Player Experience Questions for Eliminate.

<table>
<thead>
<tr>
<th>Question</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aiming</td>
<td>Aiming at the other player was easy</td>
</tr>
<tr>
<td>GameLag</td>
<td>The game felt delayed or laggy</td>
</tr>
<tr>
<td>OpponentLag</td>
<td>The other player seemed delayed</td>
</tr>
<tr>
<td>Controls</td>
<td>The mouse and keyboard controls worked well</td>
</tr>
<tr>
<td>DelayIssues</td>
<td>There were no issues with delay in this game</td>
</tr>
<tr>
<td>Improvement</td>
<td>I was getting better at shooting/aiming throughout the game</td>
</tr>
</tbody>
</table>

at the other player was easy.” (Z=-2.62, p=0.009), “The game felt delayed or laggy.” (Z=-2.02, p=0.044) and “There were no issues with delay in this game” (Z=-2.26, p=0.024). In all three cases, participant responses favoured the Shooter condition. Following the game trials, participants were asked which condition they preferred. For Non-Gamers, 8 preferred the Shooter condition and 5 preferred the Server condition; for Gamers, 10 preferred Shooter and 3 preferred Server. Interview comments show that for some players, the added challenge of leading when aiming (in the Server condition) increased enjoyment as opposed to causing frustration. For example, player 13 commented, “I preferred Game A [Shooter] but compensating for the lag in Game B [Server] was actually kind of fun”; player 1 stated, “Game B [Server] is better because of the increased potential for improvement of skills and increased challenge.” This effect may have been caused by the simplicity of the shooting task, and if the game was more challenging we would not have seen players preferring the leading condition.

**Interpretation of Results**

The choice of decision perspective had a significant impact on player performance, with score and accuracy being significantly better in the Shooter condition. This is not surprising, as in this condition, the player scores a hit whenever he shoots with his cross-hairs over the enemy’s avatar.

This performance difference was seen by both Gamers and Non-Gamers. Surprisingly, score and accuracy measures for the Gamers were not significantly better than for the Non-Gamers. We attribute this to the fact that the Gamers were frequently paired with other Gamers who were also better at avoiding being hit. When we analyzed separately the five trials in which Gamers were paired with Non-Gamers, we did find a significant effect of Expertise on both score (F\textsubscript{1,8} =13.98, p=0.002) and accuracy (F\textsubscript{1,8} =14.77, p=0.001) with Gamers performing significantly better than the Non-Gamers.
Gamers were more aware of the effects of latency. For the ‘game lag’ question, they reported that the Server condition felt significantly more laggy and for the ‘delay issues’ question they reported that the Shooter condition had fewer issues with delay. Non-Gamers responses for these questions showed no significant difference between the conditions. However, despite not subjectively seeing a difference, Non-Gamers’ performance was affected by the choice of technique. This suggests that Non-Gamers may not notice subtle differences in consistency maintenance techniques because they are too busy with general gameplay, but that these differences may nevertheless impact their performance. A main lesson is that consistency between players’ views may be less important than perceived local correctness (i.e., intuitive resolution of game-critical events).

5.2.3 Q3: How do different error repair strategies (warping or smooth correction) change experience?

**Player Performance**

The Speed Daemons game was used to examine the difference between three error repair techniques: instantaneous correction (Warping), interpolated correction (Smooth), and a technique that avoided all corrections of the remote car (Local Only). We analyzed the effects of Correction Method on individual performance (game score and puddle hits). ANOVA did not show a main effect of Correction Method on either player score ($F_{2,72}=0.20$, $p=0.82$) or puddle hits ($F_{2,72}=0.29$, $p=0.75$) and there was no interaction between Correction Method and Expertise on either measure (score: $F_{2,72}=0.02$, $p=0.98$; puddle hits: $F_{2,72}=0.02$, $p=0.98$). ANOVA did show a main effect of Expertise on both score ($F_{1,72} =5.12$, $p=0.027$) and puddle hits ($F_{1,72} =7.62$, $p=0.007$). As shown in Figure 5.10, the score was higher for Gamers and the number of puddle hits was lower.

**Player Experience**

For the experience questions (Figure 5.11), Friedman tests showed a main effect of Correction
Chapter 5. User Study: Consistency Maintenance Design Space

Figure 5.10: Speed Daemons: Performance (means) by expertise (Gamer / Non-Gamer) and by correction method

Figure 5.11: Speed Daemons: Questionnaire median responses by correction method

Figure 5.12: Speed Daemons: Questionnaire median responses by expertise (Gamer / Non-Gamer) and by correction method

Method for the last three of the four questions (see Table 5.3). Since Friedman tests do not determine interactions, we divided the data by Expertise, and carried out secondary analyses with the Gamer and Non-Gamer data (Figure 5.12).
Chapter 5. User Study: Consistency Maintenance Design Space

For the Gamers, Friedman tests showed a main effect of Correction Method for all of the last three questions: for the ‘my jump’ question, ($\chi^2=7.37$, $p=0.025$); for the ‘delay issues’ question, ($\chi^2=10.3$, $p=0.006$); and for the ‘overall’ question, ($\chi^2=13.0$, $p=0.002$). Follow-up Wilcoxon tests showed that for the ‘delay issues’ question, Warping was significantly worse than both Smooth and Local Only and that Smooth was significantly worse than Local Only ($p < 0.05$). For the ‘overall’ question, both Warping and Smooth were significantly worse than Local Only ($p < 0.05$). For Non-Gamers, Friedman tests only showed a main effect of Correction Method ($\chi^2=8.19$, $p=0.017$), for the third question. “There were no issues with delay in this game”. Follow up Wilcoxon tests showed that both Warping and Smooth were worse than Local Only ($p < 0.05$). Following the trials, the participants indicated their overall preferences. Among Non-Gamers, participants were evenly split (each technique was chosen by four people, with one participant stating no preference). Among Gamers, three people preferred Warping, two preferred Smooth, seven preferred Local Only, and one did not state a preference.

Interpretation of Results

Surprisingly, smooth corrections provided only a marginally better player experience than warping. (Gamers found there to be more issues with delay in warping than smooth corrections, but otherwise there were no significant differences between the two.) We attribute this to the large magnitude of corrections in Speed Daemons and to the very visible nature of the corrections. Corrections occurred when the cars collided with each other and the local client predicted that they did not, or vice versa. When a correction occurred, it was possible for the cars to be on the wrong sides of each other. To do the correction smoothly, the cars quickly drove around each other. Participants’ lukewarm response to smooth corrections suggests that these large, rapid movements may be as problematic as instant warping. In the Local Only condition (which used Remote Lag), the position of the remote car was always correct, just delayed in time. The local client can still make incorrect predictions about whether the local car collided with the remote car, and thus there may still be corrections to the local car when the true remote position arrives over the network. However, these corrections were smaller and less frequent than with dead reckoning, and had less impact on player experience. As with Eliminate, gamers and non-gamers had similar experiences with the different techniques, although gamers continued to be more sensitive to the differences.
The clear winner for player experience is Remote Lag, which reduced the number and magnitude of corrections. Remote Lag leads to greater divergence in state among the participants, but generally leads to smoother animation. This is another indication that factors other than overall consistency can be critical to player experience.

5.3 Summary of Results

To summarize, the following are the important lessons from the study:

- The algorithm that leads to the best player performance does not always lead to the best player experience. In both Paddle Blasters and Eliminate, we saw a decoupling of player performance and experience. In Paddle Blasters, medium consistency was chosen by only 4 of 26 players as their preferred algorithm, yet delivered the best score (in a statistical tie with low consistency). While most players in Eliminate preferred the direct aiming that allowed them to score more hits, some preferred the leading version that provided more challenge.
- Similarly, the algorithm providing the best consistency does not always lead to the best player performance or the best player experience. In Paddle Blaster, the high consistency condition led to lowest scores and worst experience.
- In all three games, we saw that expert gamers are more likely to perceive anomalous behaviour due to latency than are novice players. Surprisingly, novice players can fail to distinguish negative behaviour even when their performance suffers from it. This was most evident in Eliminate where the novice players had significantly worse accuracy in aiming in the Server condition, but reported no significant difference when responding to the experience questions.
- Smooth corrections are not always better than warping, particularly for large corrections that may be required after collisions between entities. In Paddle Blasters, the participants reacted negatively to the small warps in position that caused jerky motion. However, in Speed Daemons, smooth corrections were found to be only marginally better than warping. The large corrections where a car moved quickly around the other car were disconcerting, even when they were done smoothly.

We next discuss the conclusions we can draw from these findings.
5.4 Discussion

Our study shows that the consistency maintenance problem is multi-dimensional, as captured in the three dimensions of our design space. The three example games show that focusing on consistency alone is insufficient, and that the factors of decision-making and error repair are important in designing a complete consistency maintenance scheme. The study indicates that designers must consider questions such as the following:

• How important is consistency to decision-making? For example, in a scenario where two players attempt to pick up the same “loot” at the same time, it is critical that all clients make the same decision, and a consistency maintenance scheme with a high degree of consistency, such as local lag, should be adopted.

• Can decision-making be viewed as asymmetric, in the sense that the results of the decision are more critical to one player than another? If so, then the remote lag consistency maintenance scheme can be used.

• To what degree can corrections be tolerated? For example, in games where player movement is highly predictable, dead reckoning schemes may be appropriate as they will result in few corrections, and these can be easily and (often) imperceptibly repaired. However, if the corrections are highly noticeable, remote lag and local lag, both of which reduce the number and size of corrections, may be more suitable.

The multi-dimensional nature of the consistency maintenance problem highlights the importance of user testing to determine the true effects of algorithmic choices on player experience. The metrics we presented in Chapter 4, allowed us to quantitatively measure the effects of the different consistency maintenance algorithms. This is important for understanding the effects of the algorithms, but the metrics cannot be translated directly into player experience. It is only through actual user testing that we can determine the true impact of different consistency maintenance choices on player experience. As we have seen in this study, the best player experience is not always provided by the algorithm that leads to the highest consistency, or even the best player scores. We saw two reasons why the participants actually preferred games where they scored worse. First, part of the fun of playing a game is the challenge of trying to accomplish a goal. If the game is too
easy, players may find it boring and not as fun as a more challenging version. For this reason, we believe some of the gamers enjoyed the version of Eliminate where they could not aim directly at their opponent, but instead had to determine how much to lead them. Second, in one condition of Paddle Blasters, warps were used to correct the canoe position as quickly as possible which allowed the players to score better than in the other two conditions. However, the warps lead to jerky animation that many players found to be visually annoying.

Play testing also showed the fallacy of assuming that smooth corrections would always provide a superior experience over warping. Testing showed that remote lag was a surprisingly good option in many situations, as it negated the need for most of the corrections.

The study shows that the type of game and the target player group are important considerations in developing a consistency maintenance strategy. Games designed for casual players may be able to use a simpler strategy, as many of the subtle effects may go unnoticed by the players. However, fast-paced games designed for expert gamers must carefully consider the impact of consistency maintenance choices on each interaction.

We have seen that there are interesting interactions between the three axes of lag compensation technique, decision-making and error repair. The use of remote lag can reduce the number of corrections, making it easier to provide a smooth animation. However, remote lag also leads to the greatest state divergence and can sometimes make it more difficult to make game-critical decisions that appear intuitive to all players. In cases where there is an asymmetric relationship between the players involved in the decision, as in the shooting example described previously, remote lag can work well. However, when there is a symmetric relationship such as in a racing game, it is possible for each player to believe that they are in the lead, when from the server perspective they are behind the other players.

Dead reckoning has the potential to help with this and works well when the movements of the game entities are highly predictable, such as the forward motion of the cars in a race. However, when the entity erratically stops and starts and changes direction, a large number of jarring corrections can result. Smooth correction techniques can mask the negative effects of the error repairs, but when the game uses physics and entities collide with each other, the corrections may be too large to mask with smooth corrections. Local lag eliminates the problems of decision-making and error
Chapter 5. User Study: Consistency Maintenance Design Space

repair, but is only suitable for slow paced games.

In a perfect world, game developers would prefer complete consistency among all players’ views of the game world. However, as a consequence of the tight performance requirements of most networked games, we must accept a degree of inconsistency and sometimes even make incorrect decisions. Ultimately, the key to a good consistency maintenance strategy is providing players with an intuitive local view of the game world. Similar to the findings of Greenberg and Marwood [50] for some classes of real time groupware, it is not necessary that each player has an identical view of the state, merely that the views are consistent enough and that the outcomes of game-critical decisions can be rationalized by the players.

In the next chapter, we follow-up on our findings related to smooth corrections, by investigating the range of techniques for error correction and the factors that affect player reaction to corrections.
Chapter 6

User Study: Repairing Inconsistencies

In the previous chapter, we looked at consistency maintenance as a multi-dimensional problem. We showed that it is insufficient to focus on consistency alone, and that the factors of decision-making and error repair must also be considered in designing a complete consistency maintenance scheme. In this chapter, we now take a deeper look into the third axis of our design space, the problem of error repair.

Despite the large body of research on techniques for maintaining consistency in networked games, the issue of how to repair inconsistencies when they are detected has been largely overlooked. As we first discussed in Chapter 3, inconsistencies can be repaired immediately, possibly resulting in a confusing or jarring experience, or they can be gradually corrected over time. There is little guidance available as to which of these broad approaches best reduces the impact on player experience. To address this, we conducted an experiment that provides answers to three broad questions:

- How noticeable and annoying are corrections in networked games?
- How do different gameplay conditions affect a player’s reaction to these corrections? and
- What techniques may make corrections less annoying to game players?

In this chapter, we present a study that tested 18 people in five variations of a custom game that implemented several techniques for error correction. For each correction, we recorded whether or not players noticed the correction, and if they did notice it, the degree to which they found the correction annoying. Some of the results were largely expected: corrections are most problematic...
Chapter 6. User Study: Repairing Inconsistencies

when they occur within the player’s locus of attention, and corrections are far more annoying when they impact the game situation. Other results were surprising. Smoothing corrections over even a 250 ms interval significantly decreased how frequently corrections are noticed. Performing corrections with a gradually changing velocity is surprisingly better than using constant velocity. And the likelihood that a player notices a correction depends more on the correction’s velocity than its magnitude.

6.1 Background

Techniques for repairing inconsistencies were first introduced in Chapter 3. Before delving into our study, we briefly review these techniques.

Inconsistencies arise due to a combination of network latency and the use of prediction by the client to determine the local game state. Because of the real-time performance requirements of many networked games, some inconsistency must typically be tolerated in exchange for faster feedback to the player and quicker resolution of game events. However, these inconsistencies can have a negative impact on player performance and experience [8, 69] such as the confusion caused by sudden warps in position, the frustration of shooting directly at another player and missing [9], or the perplexity of suffering damage from a grenade while protected by an invincibility shield [2].

When an inconsistency is identified (e.g., when a prediction algorithm has moved an avatar to the wrong location), the error must be repaired – and since errors often involve a visible object in the game, the repair must be visually understandable. As we describe in more detail below, some techniques provide smoother corrections (e.g., interpolating between the incorrect and correct locations over a time period) but at the cost of prolonging the inconsistency, in that it takes longer for the two nodes to reach the same state.

To get a better understanding of inconsistencies and how they can be repaired, let’s consider two players, Alice and Bob. Alice is controlling her avatar. She is moving from left to right and then suddenly turns upward. We see this motion in Figure 6.1a. Bob is playing on another computer and there is network latency between their two computers. In Figures 6.1b, c and d, we see three possible representations of what might be shown on Bob’s screen. In all cases, Bob sees Alice’s
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.1: The dashed line (a) shows the motion of a game object that is travelling from left to right and then sharply turns upward. The next three images show the motion of that same object as viewed on a remote display when using (b) warping, (c) simple smooth corrections and (d) smooth corrections with a gradually adjusted correction rate.

avatar moving from left to right, but when she turns, his computer does not instantly receive that information. So his computer continues to predict that Alice is still moving to the right. When the update arrives at Bob’s computer, there are three options. He could see Alice’s avatar immediately warp to a new predicted position (Figure 6.1b). He could see Alice’s avatar speed up to join her newly predicted path (Figure 6.1c). Or, he could see Alice’s avatar’s speed change more gradually to join her newly predicted path (Figure 6.1d).

There are significant tradeoffs between these solutions, and to date, their impact on players’ experience has received very little attention. Correcting immediately may affect players’ immersion in the game world since a player’s attention will be drawn toward any sudden discontinuities in the position or motion of objects in the game [69]. A player may also lose context, for example, if an avatar that was in front of him is suddenly behind him and now out of his field of view. Smooth corrections may reduce the jarring effect of corrections [91]; however, they prolong the inconsistency, which may be unacceptable in some game situations. Fiedler [42] suggests the rule of thumb of moving 10% toward the true position during each frame, but if the correction is large, simply warping to the new position. More sophisticated forms of smooth correction are possible; e.g. varying priorities may be given to different objects in the game to ensure the objects of higher
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.2: In the Space Rock-it game, players (a) observed a single space ship, (b) observed two space ships and asteroids, (c) coloured the asteroids by bumping into them or (d) attempted to shoot the opposing ship.

importance deviate the least from their true position [16]. To date, there has been little effort to characterize when smooth corrections are effective, and how they should be applied.

6.2 Error Repair Study

As indicated previously, the research described in this chapter addressed three broad questions: how noticeable and annoying corrections are in networked games, how different gameplay conditions affect players’ reactions to corrections, and what techniques make corrections less annoying for game players.

We addressed these questions by conducting a user study using a custom built game that allowed us to inject errors and corrections into the game and to obtain player feedback. The game, Space
Chapter 6. User Study: Repairing Inconsistencies

Table 6.1: Summary of Game Conditions: The game conditions differed based on the task the participants were performing, the number of entities displayed on the screen and the relationship of the participant to the object being corrected.

<table>
<thead>
<tr>
<th>Condition Name</th>
<th>Task</th>
<th>Game Entities</th>
<th>Ship Corrected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observing - Single</td>
<td>Observe Only</td>
<td>One</td>
<td>Observed Ship</td>
</tr>
<tr>
<td>Observing - Many</td>
<td>Observe Only</td>
<td>Many</td>
<td>Observed Ship</td>
</tr>
<tr>
<td>Colouring</td>
<td>Colour Asteroids</td>
<td>Many</td>
<td>Other Ship</td>
</tr>
<tr>
<td>Controlling</td>
<td>Colour Asteroids</td>
<td>Many</td>
<td>My Ship</td>
</tr>
<tr>
<td>Shooting</td>
<td>Shoot Other Ship</td>
<td>Many</td>
<td>Other Ship</td>
</tr>
</tbody>
</table>

Rock-it, as shown in Figure 6.2, featured two space ships and asteroids moving about in a 2-D world. The Space Rock-it game was divided into five separate game conditions in which the participants either observed the motion of the objects on the screen or controlled the motion of one of the space ships. We now describe how the game was designed in order to answer our research questions.

6.2.1 Q1: How Noticeable and Annoying are Corrections?

As participants played the Space Rock-it game, errors of differing magnitude were randomly injected into the game and then corrected using a variety of algorithms. Players were instructed to press the space bar when they saw a correction, and then rate how annoying they found the error. This allowed assessment of what magnitude of errors people perceive, under which conditions, and how annoying they found them. This procedure is detailed below.

6.2.2 Q2: How Do Gameplay Conditions Affect Reactions to Corrections?

In addition to the question of how annoying people find corrections to be in general, we wished to investigate the impact of three key factors on how players notice and react to corrections. These factors are distraction, locus of attention, and involvement with gameplay. First, the amount of activity and number of objects or clutter in the game interface may distract players and affect their ability to notice corrections. Second, at any given time, people focus on a single area of the display, and the proximity of a correction to this locus of attention may affect how likely players are to notice it. Finally, the very act of playing a game may affect a player’s ability to notice corrections:
playing a game may either distract players, causing them to not notice the corrections, or it may exacerbate the effect of a correction if it breaks immersion or affects gameplay performance.

To test these hypotheses we created five game conditions within the Space Rock-it game (see Table 6.1). The rationale behind each condition is discussed below.

First, to investigate whether or not a visually busy game might be able to mask corrections, we required two conditions in which the participants would be performing the identical task. However, in one condition, there would be as little as possible to distract the participants and in the other condition there would be more objects on the screen that might distract them. For this, we chose the task of simply observing the screen. In one condition, *Observing - Single* (Figure 6.2a), the participants observed the motion of a single pink space ship and corrections were applied to the ship at regular, but random, intervals. In the other condition, *Observing - Many* (Figure 6.2b), the participants again observed the motion of the pink space ship; however, there was also a second green ship and 40 asteroids moving about on the screen. Comparing the number of corrections detected by the participants in these two conditions would allow us to see if the distractions had any effect.

To investigate the effect of the participant’s locus of attention, we created conditions requiring the participants to pay varying degrees of attention to the object being corrected. In the *Observing* conditions described above, the participants were asked to observe the motion of the space ship which was being corrected, and thus, all of their attention was focused on it. By adding conditions involving game play to the study, we manipulated the degree to which the participants were able to focus on the object being corrected. In the “Colouring Game” (Figure 6.2c), the players tried to change the colour of as many of the asteroids as possible. The colour of an asteroid was changed by bumping into it with the space ship. At the start of the game, all of the asteroids were yellow in colour. If the green ship hit an asteroid, it turned green. If the pink ship hit an asteroid, it turned pink. This game was used to create two conditions. In the first of these, the *Controlling* condition, the corrections all occurred to the space ship being controlled by the player, where we presumed the player would be focusing most of her attention. In the second condition, *Colouring*, the corrections all occurred to the non-player controlled ship where we expected the player would be paying much less attention. We also wanted a condition where the amount of attention paid by the player to the
Chapter 6. User Study: Repairing Inconsistencies

object being corrected fell between these two conditions. In the “Shooting Game” (Figure 6.2d), the player controlled the green ship which was firing missiles, and the player attempted to steer the ship to aim the missiles at the other ship. In this case, the corrections all occurred to the non-player controlled (pink) ship. We intended that in this condition, the player would divide her attention between the ship she was controlling and the one at which she was aiming. By comparing the number of corrections detected by the participants in each of these conditions, we would be able to determine the degree to which the player’s locus of attention affected their ability to notice corrections.

To investigate how playing a game impacted a player’s opinion of corrections, we used two conditions where we believed that the players would be paying equal attention to the object being corrected, but in one case the participants would be playing a game, and in the other they would not. We already had two such conditions in our study, the Observing - Many condition where the players were observing the space ships and asteroids and the Controlling condition where the players were controlling the ship being corrected in the “Colouring Game”. We could thus determine how the act of playing a game affected player perceptions by comparing number of corrections detected and the player ratings of the corrections for these two conditions.

6.2.3 Q3: How Do Error Repair Techniques Impact Player Experience?

Our third major question addresses the effect of different repair techniques on player experience. We implemented three repair strategies in the Space Rock-it game: warping, smooth corrections with a constant correction rate, and smooth corrections with a variable correction rate. To provide guidelines for repairing inconsistencies, we compared warping corrections to smooth correction and we compared smooth corrections with a constant correction rate to those with a gradually adjusted rate. We also calculated detection limits for the various corrections based on the time interval over which the correction was performed.

When performing smooth corrections, the simplest approach is to pick a target location where the game entity should arrive at some point in the future and then adjust the entity’s velocity to smoothly move to that location (Figure 6.1c). The problem with this approach is that, although there are no instantaneous jumps in the position of the entity, there is a sharp and sudden change in
its velocity. Smed and Hakonen have suggested that gradually changing the velocity (Figure 6.1d) would make the correction less noticeable and thus be preferable [91]. We sought to confirm this by implementing two versions of smooth corrections. In the first version, we set a time interval (ranging from 250 ms to 1000 ms; refer to the Procedure section of Section 6.2.4 for specific values) over which the correction is to be completed and then move the entity toward the correct location at a constant rate. This is shown by the solid blue line in Figure 6.3. In the second version, we perform the correction over the same time interval, but initially move the entity slowly, increasing the correction rate over time as shown by the dashed green line in Figure 6.3. The correction rate increases linearly up to the mid point of the correction and then decreases linearly. In both cases, the correction is completed over exactly the same time interval and the average correction rates are identical. Note, however, that when the correction rate is changed gradually, the correction rate is actually higher in the middle of the correction. We compared the number of corrections detected and the player ratings for these two forms of smooth correction.

We expected that if smooth corrections are performed increasingly slowly, there eventually comes a point where the corrections would be undetectable. In order to compute the detection limits, we included a range of correction sizes and correction times in the study. The specific values are listed in the Procedure section of Section 6.2.4.
In the following section, we now describe the study in more detail.

6.2.4 Method

Eighteen participants (16 male and 2 female) were recruited from the Queen’s University student population. All of the participants were regular game players who played fast-paced action games for a minimum of four hours per week. For this study, we specifically recruited regular game players, as our previous study (described in Chapter 5) showed that non-gamers had significantly less ability to detect anomalies in games.

Procedure

The participants played the game seated at a computer with a 27 inch monitor and used the keyboard both to control their on screen avatar (a space ship) and to provide feedback when they observed a correction. Although the study investigated repairs to inconsistencies in networked games, we used a single player game in which errors and the subsequent corrections were synthetically generated. This allowed us to carefully control the frequency and magnitude of repairs to inconsistencies.
Chapter 6. User Study: Repairing Inconsistencies

The study began with a brief training session in which the participants observed the normal motion of the space ship for about thirty seconds. The ship then suddenly warped to a new position and the participants were asked to press the space bar when they saw this happen. At this point, the rating system (Figure 6.4) appeared on the screen and was explained to them. The participants continued to observe the space ship for an additional minute and pressed the space bar each time they noticed a correction.

Following this training period, the participants played the five different game conditions listed in Table 6.1, with each condition lasting approximately eight minutes. The conditions were always played in the same order:

- **Observing - Single**: Observing one ship
- **Observing - Many**: Observing two ships and asteroids
- **Colouring**: Colouring game with corrections occurring to the other ship
- **Controlling**: Colouring game with corrections occurring to the player controlled ship
- **Shooting**: Shooting game with corrections occurring to the target ship

We did not randomize the order of the conditions, because we wanted the players to become thoroughly familiar with the motion of the space ships and asteroids before beginning the game play conditions. The results indicate that any learning effect had little impact on the results, as we found no significant difference in the total number of corrections detected between the **Observing - Single**, **Observing - Many** and **Controlling** (first, second and fourth) conditions. This is discussed in more detail in Section 6.3.

During each condition, corrections occurred to the position of one of the two space ships. The players were asked to indicate when they noticed a correction by pressing the space bar. As soon as they pressed the space bar, the game paused and the participants rated how annoying they would have found a similar correction had it occurred in an actual game (Figure 6.4). The rating scale was based on the **Mean Opinion Score** (MOS) [68], a metric established for subjective quality measurements of audio and video signals. The MOS is based on human perception and has values ranging from 1 to 5 with a 1 indicating the poorest quality and a 5 indicating no signal degradation. This scale has been adapted and used by several researchers [32, 100, 104] for measuring video game quality, particularity when affected by network latency.
Chapter 6. User Study: Repairing Inconsistencies

The participants also had the option of pressing the delete key to indicate that they had accidentally pressed the space bar. While the game was paused, the positions of the space ships and the asteroids were dimly visible in the background behind the text. This enabled the players to remain focused on the game while providing feedback about the corrections. The game resumed immediately after the player pressed a number key to rate the correction. The order of the errors was randomized and intervals between the errors were uniformly distributed over three to eight seconds.

Following the study, we conducted a semi-structured interview with each participant to elicit additional information about how they rated the corrections and the types of corrections they found most annoying.

During each condition, there were 60 corrections. We varied the corrections along three axes:
- the magnitude of the correction,
- the correction time – the time over which the error was repaired,
- the smoothness of the correction

Each of these axes is described in more detail below.

**Magnitude of the Correction**

The magnitude of a correction is measured in centimetres (cm) on the display. The space ship was 4 cm long by 1.5 cm wide and travelled at a speed of 12 cm/s. This size was chosen based on aesthetics and the requirement that the ship be easily visible on the screen. The speed was selected to make the ship comfortable to control. In setting the magnitude of corrections, we considered the range of errors that might typically be found with modern network conditions and based the size of the correction on how far the ship could travel in 125, 250 and 500 ms. With a ship velocity of 12 cm/s, the correction magnitudes were thus 1.5, 3.0 and 6.0 cm respectively.

**Duration of the Correction**

When repairing inconsistencies, a game developer must choose between making the repair instantly with possibly jarring effects or prolonging the inconsistency and progressively fixing it over time. While it seems intuitive that repairing inconsistencies over time would be superior to warping, this had never been tested prior to this study, and there was no experimental data to indicate how quickly the repairs should actually be made. We performed a pilot study to determine repair times.
that ranged from very obvious to rarely detected, and based on these, we selected five duration values ranging from 0 to 1 second: 0 ms (warping), 250 ms, 500 ms, 750 ms and 1000 ms.

Smoothness of the Correction

When repairing inconsistencies using smooth corrections, the simplest solution is to adjust the speed of the entity and move at a constant rate toward the new position as shown in Figure 6.1c. This has the potential to be slightly jarring for the player as the entity instantaneously changes speed and direction. As shown in Figure 6.1d, “ smoother” corrections gradually adjust the correction rate so that there are no sudden changes in velocity [91]. The actual correction rates for these two techniques is shown in Figure 6.3. When the correction rate is changed gradually, the actual rate is faster at the midpoint of the correction.

In our study, we compared these two options of constant and variable correction rates.

Data Collection and Analysis

The Space Rock-it game was used to log data about the corrections and the participant responses. For each correction, we recorded:

- Date and time
- Game condition
- Correction type (constant or variable correction rate)
- Magnitude of correction
- Correction duration
- Positions of the ships
- Participant response, or non-response if they did not detect the correction.

We also recorded if the participants had indicated that they had seen a correction when none had occurred.

We analyzed the data using one-way ANOVAs to test for effects on the number of corrections observed. Player opinion of the corrections was analyzed with Wilcoxon tests on medians.
Chapter 6. User Study: Repairing Inconsistencies

Table 6.2: Summary of participant responses

<table>
<thead>
<tr>
<th>Description</th>
<th>Number Reported</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Very Annoying</td>
<td>579</td>
<td>1</td>
</tr>
<tr>
<td>2 Annoying</td>
<td>660</td>
<td>5</td>
</tr>
<tr>
<td>3 Slightly Annoying</td>
<td>760</td>
<td>2</td>
</tr>
<tr>
<td>4 Perceptible but not Annoying</td>
<td>694</td>
<td>19</td>
</tr>
<tr>
<td>5 Imperceptible</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Not detected</td>
<td>2,704</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>5,400</strong></td>
<td><strong>27</strong></td>
</tr>
</tbody>
</table>

6.3 Results

The results of our study allow us to address the three broad questions posed earlier in the paper: how noticeable and annoying do players find corrections; how do gameplay conditions affect reactions to corrections; and how do error repair techniques impact player experience?

In the study, there were a total of 5,400 corrections over all the different conditions, of which the participants noticed 2,696 or just under 50%. Table 6.2 summarizes the participant responses based on the rating they assigned to each correction. There were a large number of reports at each level, indicating that participants used all available categories. We anticipated that none of the participants would rate any of the corrections as a 5, “Imperceptible”, because if a correction was imperceptible, they should not have pressed the space bar. In fact, one participant did rate three of the corrections as “Imperceptible”. During the post-experiment interview, he explained the response stating that he thought maybe a correction was occurring, but was not sure, so he rated it as “Imperceptible”.

We also collected data to measure the number of times a participant pressed the space bar when no correction had occurred (false positives). There were 27 false positives or 0.5% of the total corrections. We believe this to be a sufficiently small value as to not impact the overall results of the study and thus the false positives were not considered in the subsequent data analysis.
Chapter 6. User Study: Repairing Inconsistencies

6.3.1 Q1: How Noticeable and Annoying are Corrections?

Participants noticed 50% of the corrections that occurred during gameplay, and rated 46% of the corrections they noticed as either “Annoying” or “Very Annoying”. This is consistent with the anecdotal experience (described in Section 1.1) of games such as Battlefield 4 and Rainbow Six: Lockdown, where consistency issues were reported by players and reviewers as significantly negatively impacting player experience. This indicates that inconsistencies in video games are a real problem, and that corrections can be both noticeable and annoying for game players.

6.3.2 Q2: How Do Gameplay Conditions Affect Reactions to Corrections?

While the data from Table 6.2 shows that corrections are frequently noticed and are frequently considered annoying, we are primarily interested in what specific factors lead to corrections being more or less visible or annoying. We investigated the impact of three key factors on how players notice and react to corrections. These factors are distraction, locus of attention, and involvement with gameplay.

Distractions

We used the two Observing conditions to evaluate how placing additional elements in the scene might affect the players’ abilities to notice when corrections occurred. In both these conditions, the players were merely observing the motion occurring in the scene. In the first condition, Observing - Single (Figure 6.2a), there was a single pink space ship travelling around the screen. In the second condition, Observing - Many (Figure 6.2b), there were two space ships and forty asteroids moving around. In both conditions, the participants were asked to focus their attention on the pink ship, and all corrections were applied to the position of the pink ship.

In each condition, there were a total of 60 corrections. In the Observing - Single condition, the participants noticed on average 35 of the corrections compared to 36 in the Observing - Many condition (see Figure 6.5). A paired-samples t-test showed that the addition of the second space ship and the asteroids did not show a significant effect on the number of corrections that were detected ($t(17)=0.98$, $p=0.342$). Because we did not find a significant difference between these two

101
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.5: The addition of distractions did not show a significant effect on the number of corrections observed. The error bars shown represent the standard error for the number of corrections detected.

Figure 6.6: The participants noticed more corrections when their locus of attention was directly on the game entity to which the correction was applied. The bars show the average number of corrections noticed in each condition.

conditions, we will only present the Observing - Many condition in all further discussion of the results, and this condition will be referred to simply as Observing.
Chapter 6. User Study: Repairing Inconsistencies

**Locus of Attention**

In a game, a player’s “locus of attention” is the point on the screen where the player’s visual attention is focused. In both the *Observing* and *Controlling* conditions, the player’s attention was focused directly on the ship to which the corrections were being applied. However, in the *Shooting* condition, the player was forced to divide her attention between the ship she was shooting at and the ship she was controlling. In the *Colouring* condition, the corrections all occurred to the AI controlled ship and the player focused very little attention on the motion of that ship. The number of corrections detected in each of these conditions as well as the standard error is shown in Figure 6.6.

One-way ANOVA showed a significant effect of the locus of attention on the number of corrections observed ($F(3,51)=48.17$ $p<0.001$). Follow-up pairwise t-tests using Bonferroni correction indicated that the number of corrections observed in the *Colouring* condition was significantly less than all the other conditions ($p<0.001$) and that the *Shooting* condition was significantly less than both the *Observing* ($p=0.001$) and the *Controlling* ($p=0.007$) conditions. No significant difference was found in the number of corrections observed between the *Observing* and *Controlling* conditions ($p=0.190$). The participants therefore saw most corrections in the cases where they were focused on the avatar being corrected, less in the split-attention case, and less yet when attention was focused on another avatar.

To investigate more deeply peoples’ ability to perceive corrections while focused on another avatar, we performed further analysis on the *Colouring* condition, where players control the green ship that is not being corrected, while corrections are applied to a pink ship. We hypothesized that players would be more likely to notice corrections when the green ship that they control is close to the pink ship that is being corrected. We tested for a correlation between the distance between the two ships and the likelihood of observing corrections. We coded each correction as a 1 if it was detected and a 0 if it was not detected and correlated versus distance between the two ships. We found a weak negative correlation between distance and observation of corrections, ($r(1078)=-0.134$, $p<0.01$), establishing that indeed players might be more likely to observe a correction when it occurred near to their locus of attention. The regression line is shown in Figure 6.7.
Involvement with Gameplay

In both the *Observing* and *Controlling* conditions, the player’s attention was focused directly on the ship to which the corrections were being applied. As described in the previous section, we found no significant difference between the number of corrections detected by the participants in each of these conditions. However, in the *Controlling* condition when a correction occurred it often interfered with the player’s ability to touch an asteroid. We hypothesized that this would cause players to rate the corrections as more annoying than in the *Observing* condition. A Wilcoxon signed rank test indicated that the corrections were rated as significantly more annoying in the *Controlling* condition than in the *Observing* condition ($z = -4.84$, $p<0.001$). Figure 6.8 shows the correction ratings for the *Observing* and *Controlling* conditions. More corrections were rated as “Annoying” or “Very Annoying” in the *Controlling* condition and more corrections were rated as “Not Annoying” in the *Observing* condition.

6.3.3 Q3: How Do Error Repair Techniques Impact Player Experience?

We now compare the three principle correction techniques of warping, smooth corrections with a constant correction rate and smooth corrections with a variable correction rate and calculate detection limits for the corrections based on the time interval over which the correction was performed.
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.8: The participants rated the corrections as significantly more annoying when they affected the game play.

We begin by comparing warping to smooth corrections.

**Smooth Corrections Versus Warping**

In our study, smooth corrections were performed over four time periods ranging from 250 ms to 1000 ms. We first consider the shortest repair period for smooth corrections, 250 ms. As shown in Figure 6.9, in all the conditions, fewer corrections were detected when they were performed smoothly over 250 ms than if warping was used. In all cases, the difference was significant (p < 0.001).

In addition to comparing the number of corrections detected, we also compared the participants’ ratings of the corrections. For any correction the participants did not detect, we assigned a rating of 5, “Imperceptible”. As shown in Figure 6.10, in all the conditions, the smooth (250 ms) corrections were rated as less annoying (shorter bar) than the warping corrections (the figure shows the median rating.) For each condition, we performed a Wilcoxon test to evaluate whether there was a difference between warping corrections and the corrections performed over a 250 ms window. In each case, the results indicated a significant difference: *Observing* (z = -11.61, p < 0.001), *Controlling* (z = -9.72, p < 0.001), *Shooting* (z = -9.92, p < 0.001) and *Colouring* (z = -8.30, p < 0.001). This indicates that players found smooth corrections to be less annoying than warping in all four game conditions,
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.9: In all conditions, more corrections were noticed when warping was used.

Figure 6.10: Corrections were rated on a scale of 1 (Very Annoying) to 5 (Imperceptible). A taller bar indicates that the corrections were more annoying.

when smooth corrections were made over a 250 ms window.

Similar comparisons over longer correction windows (500 ms to 1,000 ms) yielded the same results, that in all cases, smooth corrections were preferred to warping, at the $\alpha < 0.05$ level.

Constant Versus Variable Correction Rate

To investigate the effect of the smoothness of corrections, we first looked at the number of corrections detected by the participants when the corrections were performed using a constant correction rate and when they were performed by gradually adjusting the correction rate. For each condition,
there were a maximum of twelve corrections of each type that might have been detected by the
participants: three corrections sizes (1.5, 3.0 and 6.0 cm) and four durations (0.25, 0.50, 0.75
and 1.00 ms). As shown in Figure 6.11, in each condition, fewer corrections were noticed by
the participants when the correction rate was adjusted smoothly. Paired-samples $t$ tests, however,
showed that the difference was only significant for the Controlling condition ($t(17)= 5.25, p<0.001$).
For the other conditions, the results were as follow: Observing ($t(17)= 1.23, p=0.235$); Shooting
($t(17)= 1.44, p=0.168$); and Colouring ($t(17)= 0.74, p=0.472$). When we considered the different
correction sizes, there was a significant difference between the number of corrections detected
for the small (1.5 cm) and medium (3.0 cm) corrections for both the Controlling and Shooting
conditions. For the large (6.0 cm) corrections, there was no significant difference between the
number of corrections detected, likely because these larger corrections were easier to detect and the
participants noticed in excess of 60% of the corrections in all of the conditions except Colouring.

Further analysis using Wilcoxon tests to compare the ratings of each correction indicated that
there was a significant difference in all of the conditions except Colouring. The results were as
follows: Observing ($z = -2.47, p=0.014$), Controlling ($z = -6.69, p<0.001$), Shooting ($z = -2.11,
p=0.035$) and Colouring ($z = -0.38, p=0.702$).

A review of the number of positive and negative ranks reported by the Wilcoxon test for each
correction size and correction duration were similar, indicating that this difference is applicable
over the entire range of correction sizes and durations tested in our study.

**Correction Detection Limits**

We calculated the detection limits for each correction size and test condition. We use a standard
psychophysical definition of detection limit as the level at which 50% of the participants notice the
stimulus [45]. The detection limits were measured in milliseconds. That is, we sought the minimum
correction duration in milliseconds where only 50% of the participants detected the correction. In
all cases, we noticed that the detection limit for the large (6 cm) corrections was approximately
twice as large as for the medium size (3 cm) corrections. Similarly the detection limit for the
medium size corrections was approximately twice as large as for the smallest (1.5 cm) corrections.
For this reason, we report the detection limits in terms of the correction rate (cm/s) instead of in
Chapter 6. User Study: Repairing Inconsistencies

Figure 6.11: In all conditions, fewer corrections were noticed when the correction rate was adjusted smoothly.

terms of the time over which the correction was performed and combine the detection limits for the three correction magnitudes. A correlation of the detection limits in cm/s versus the size of the correction validated this approach, showing no significant correlation for any of the conditions: *Observing* $p = 0.080$, *Controlling* $p = 0.575$, *Shooting* $p = 0.812$ and *Colouring* $p = 0.251$.

In Figure 6.12, we show the average detection limit for each condition. The *Observing* condition had the lowest detection limit ($M=5.0$, SE = 0.4), followed by the *Controlling* condition ($M=6.4$, SE = 0.8) and then the *Shooting* condition ($M=7.5$, SE = 0.7). The *Colouring* condition had a much higher detection limit ($M=26.2$, SE = 4.1). For the first three conditions, the detection limit is close to half the normal speed of the space ship in the game. In the *Colouring* condition, where the corrections occurred to the AI controlled ship and the player was paying less attention to that ship, the limit was much higher at over twice the normal ship speed.

### 6.3.4 Summary of Results

We found that the most significant factor affecting the detection of corrections was the player’s locus of attention. Distractions that did not shift the player’s focus had little impact on the number of corrections detected.

We found that for the corrections in our study, smooth corrections were vastly superior to warping. Also, based on the player ratings of the corrections, we found that it was significantly
better to make gradual adjustments to the correction rate rather than performing the correction at a constant rate.

Finally, we found that the ability of players to detect corrections is based on the speed of the correction, not its magnitude. When an entity is within the player’s locus of attention, its speed can be adjusted by approximately 50% relative to its normal speed. Significantly higher adjustments can be made when the entity is outside the locus of attention.

### 6.4 Discussion

The first objective of our study was to determine how noticeable and annoying corrections are. We found that corrections are a real problem in video games and players rated a significant proportion of them as either “Annoying” or “Very Annoying”. All of the participants rated at least some of the corrections as “Very Annoying”.

Our next objective was to determine which factors affected players’ abilities to notice corrections and the degree to which they found the corrections annoying.

We were surprised that we did not find a significant difference in the number of corrections...
detected as a result of adding distractions to the scene in our two observing conditions. In the *Observing - Single* condition, the scene only showed a single space ship and in the *Observing - Many* condition, a second ship and 40 asteroids had been added to the scene. In both conditions, the corrections all occurred to the same pink ship on which the participants were instructed to focus their attention. The additional objects all moved around the scene at a constant speed and did little to attract the participants’ attention. Grad et al. [47] have shown that users tend not to notice peripheral information unless it is flashing or animated. Thus, we believe it is likely that the objects we added to the scene did not cause the participants to shift their focus away from the pink ship. It is quite possible that, if the motion of the other ship and asteroids was discontinuous or included some other unexpected motion, the distractions might have had an effect. This tells us that in general we cannot rely on distractions to mask the effects of repairs to inconsistencies.

Interestingly, during the post-hoc interviews, 11 of the 18 participants felt that the addition of the second space ship and the asteroids did distract them. It appears, however, that most of the participants thought that they were just momentary distractions, for example participant 15 stated “Once I got distracted ’cause I was looking at the green ship…. Then, I thought wait – I shouldn’t be watching that.” and participant 3 commented “I would start wandering off and think let’s see what that green ship is doing, but I am pretty sure I didn’t miss anything.” It is also possible that the lack of difference between the two conditions might be attributed to a learning effect that compensated for the additional distraction. Although all the participants completed a short practice session prior to the study, they all watched the *Observing - Single* condition first, followed by the *Observing - Many* condition. Thus, they may have been more aware of what they were looking for in the second condition.

The most significant factor we found in determining a player’s ability to notice corrections was where they were focusing their attention when the correction occurred. Each of the three game-like conditions represented a different level of focus on the game object experiencing the corrections. In the *Controlling* condition, the players were controlling the space ship being corrected and their attention was almost completely focused on the point where the corrections occurred. In the *Shooting* condition, the corrections were applied to the target ship while the players were required to divide their attention between the target ship and the ship they were controlling. Finally, in
the Colouring condition, the corrections were applied to the AI controlled ship, while the position of that ship had little impact on the player. Thus, the players focused little attention on these corrections. Additionally, as the player’s attention is shifted away from the point of the correction, fewer corrections were detected (although this distance correlation was weak). Together, these observations tell us that when developing games we need to play close attention to where the player is looking and attempt to minimize the number and the visibility of corrections in that area.

In addition to knowing which corrections are most easily detected by players, it is also important to know what makes the corrections upsetting or disturbing. We found that corrections were more problematic when they interfered with the player’s progress in the game. In the Observing condition, corrections were rated as significantly less annoying than those same corrections when they occurred in the Controlling condition. Warping corrections were also found to be significantly more annoying than smooth corrections, even when the smooth corrections were completed over as little as 250 ms. In the post-hoc interviews, when the participants were asked which corrections they had found most annoying, they all described either the warping corrections or specific game situations that had affected them. Ten of the participants specifically mentioned the warping corrections; for example, participant 13 commented, “The leaping one really bothered me a lot.” and participant 9 stated “The most annoying ones were when it flashed from one point on the screen to another.” Eleven of the participants referred to specific game play situations that they found annoying; for example participant 15 stated “When I was shooting at the pink ship and tailing it and it glitched behind me” and participant 2 mentioned “The most annoying would be when you were trying to hit a ship or an asteroid and it just flies you somewhere totally different.”

This tells us that, in addition to minimizing corrections near the player’s locus of attention, developers should attempt to eliminate warping corrections and corrections that have an impact on game play. For game developers, eliminating warping corrections can usually be accomplished by replacing them with smooth corrections. Minimizing corrections that might affect game play can be more challenging. In designing an effective networking strategy, developers need to understand both where a player is focusing and which corrections might affect the game play. Then techniques can be used to to minimize, as much as possible, the need for corrections in these areas. One such technique is adaptive dead reckoning [13] in which the frequency of update messages for different
Chapter 6. User Study: Repairing Inconsistencies

game objects could be varied based on factors like where the player is focusing and whether or not the object is likely to be affected by game play. The use of remote lag [9] can almost eliminate the need for corrections to the positions of remote game entities, but adds the complexity of making game critical decisions when players have diverging views of the game world [86].

When corrections are unavoidable, they should be performed as slowly as possible using a gradual change in velocity over time. In our study we found that a correction rate equivalent to approximately half of the normal ship velocity resulted in only half of the participants detecting the correction. Further studies using different types of motion and different velocities would be required to determine the general applicability of such a guideline.
Chapter 7

Janus Toolkit and the Timelines Model

In the previous chapters, we have seen that there are a range of choices available for implementing consistency maintenance in games, and these choices can have a significant impact on players’ user experience. However, implementing and experimenting with these different algorithms can be challenging. Programming consistency maintenance algorithms necessitates the manipulation of state over time, either predicting future positions or interpolating between previous states. This requires programmers to consider not just what value shared data has, but when that value was held. Current programming languages do not provide support for dealing with time, which makes implementing consistency maintenance algorithms complex and creates a significant barrier to the invention, evaluation and deployment of such algorithms.

In this chapter, we present the timelines model, a programming model that allows programmers to deal explicitly with time. Through a series of examples, we will show how this model simplifies the implementation of a range of consistency maintenance techniques used in networked games. We also describe how the model has been implemented within the Janus toolkit and discuss our experience with the toolkit including its strengths and limitations. Janus is available open source at http://equis.cs.queensu.ca/wiki/index.php/Janus2.
Chapter 7. Janus Toolkit and the Timelines Model

Figure 7.1: Elements of the timelines model: a timeline includes a set of past and future values \((v_1, v_2, \ldots)\), along with the times at which they hold \((t_1, t_2, \ldots)\). Values between these times are computed using an interpolation function, and values after these times are estimated using an extrapolation function.

7.1 The Timelines Model

The timelines model represents shared state variables as values indexable by time. Variables represent all the values they have held in the past and all values they will hold in the future. Timelines provide:

- Get/set operations that access the timeline’s value at a given time; and
- Interpolation and extrapolation functions that estimate values for times when no value is known.

As we will show, this simple model of shared data allows the simple expression of a wide variety of consistency maintenance algorithms.

Figure 7.1 shows how the elements of the Timelines model are combined. In the example timeline, \(v_1\), \(v_2\) and \(v_3\) represent values for times \(t_1\), \(t_2\) and \(t_3\) respectively. These are known values, meaning that they have been explicitly set in the timeline.

To illustrate the model, consider \textit{fuel}, an integer timeline representing the amount of fuel in a spaceship. Storing values in the \textit{fuel} timeline is as simple as assigning those values at a specific time. Thus,

\[
\text{fuel}(-100) = 230;
\]

assigns 230 to the integer timeline at 100 ms in the past, and

\[
\text{fuel}(0) = 200;
\]
assigns 200 to the fuel timeline at the current time. (Time references are expressed in milliseconds, may be positive or negative, and are relative to the current time.)

When the value for a given time is not known, it is interpolated or extrapolated from its neighbouring known values. The value of fuel(−50) is derived as 215 using linear interpolation between the two known values of 200 and 230, and the value of fuel(100) is extrapolated to be 170. Thus, although our timeline currently contains only two known values, any value from the past or future can be determined through interpolation or extrapolation. As more known values are added to the timeline, the interpolated and extrapolated values may change.

The timelines model is fully replicated, with each client storing a local timeline object for each portion of the shared state in which it is interested. If two clients create instances of the same timeline, the timelines are automatically synchronized. Figure 7.2 shows two clients each with a copy of the same timeline. When Client 2 inserts a new value into the local timeline, the value is propagated over the network and a remote update function on Client 1 is invoked. The remote update function specifies how updates arriving over the network are to be applied. By default, the remote update function inserts the value into Client 1’s timeline. However, as we will show, it is possible to override this default behaviour to implement techniques such as smooth corrections.
Chapter 7. Janus Toolkit and the Timelines Model

To illustrate the model, we now show how timelines can be used to implement a range of consistency maintenance techniques.

7.2 Using Timelines

In this section, we illustrate the use of timelines to express a canonical set of consistency maintenance algorithms. Through these examples, we will demonstrate the features of the model, as well as its expressiveness. We begin by showing how timelines facilitate the implementation of the techniques dead reckoning, local lag and local perception filters and then show how timelines also support different frames of reference for game-critical decisions and smooth correction techniques.

7.2.1 Dead Reckoning

As discussed in Section 2.3, the key idea with dead reckoning is to predict future positions based on previous positions, and as we saw in Chapter 4, dead reckoning is an appropriate choice for games where modest state divergence and occasional corrections are considered acceptable.

Dead reckoning algorithms, however, frequently fail to account for delays caused by network latency. Typically, when a client receives a positional update, that position is set as the entity location at the current local time, not at the time the message was sent. Aggarwal et al. have shown that treating remote updates as past events (i.e., accounting for network latency) improves the accuracy of dead reckoning [1]. However, this approach requires the programming complexities of timestamping positional update messages and synchronizing the clocks on the local and remote clients.

Dead reckoning with lag awareness is built into the timelines model, requiring no extra programming by the game developer. The model’s extrapolation function provides dead reckoning as the default behaviour. When updates are received from a remote client, they are automatically inserted into the local timeline at the time they were sent, not the time they were received.

The following example shows how easily dead reckoning with lag awareness can be used in a game written with timelines. Consider that “Alice” and “Bob” each control avatars in a game. Alice’s avatar’s position is represented in a timeline of positions called alicePos. We assume the traditional game architecture where inputs are polled and the frame is rendered asynchronously.
Chapter 7. Janus Toolkit and the Timelines Model

Thus, when Alice moves to a new position \((x,y)\), the operation on Alice’s client to process the movement is simply:

\[
\text{alicePos}(0) = (x, y);
\]

This operation inserts the value \((x,y)\) into the \textit{alicePos} timeline at time 0, the current time. Then, on both Bob and Alice’s clients, when the frame is rendered, Alice’s position is drawn as:

\[
\text{DrawAvatar( alicePos(0) )};
\]

That is, the avatar is drawn at its position at the current time.

On Bob’s client, as messages indicating Alice’s movements arrive, they are automatically inserted into the timeline taking into account the network latency. For example, if the message containing Alice’s position required 60 ms to travel over the network, then the position is inserted in the \textit{alicePos} timeline on Bob’s computer at time \(-60\).

When Bob’s client uses the value of \(\text{alicePos}(0)\) to render Alice’s position, it extrapolates from her last known position. This example shows how the timelines model provides lag-aware dead reckoning as default behaviour, requiring no special programming.

7.2.2 Local Lag

With local lag, the goal is to reduce or eliminate inconsistencies by delaying local actions. As discussed in Section 2.3, the key idea behind local lag is to delay the execution of local commands long enough that the commands are able to propagate to all remote sites and can then be executed simultaneously at all locations. Adding up to 100 ms of local lag to a game can increase the degree of consistency between the views of different players, and as our studies (and those of other researchers) have shown, it is often not even noticed by the players.

Programming local lag requires mechanisms for delaying inputs and for estimating message delivery time between the different nodes. It also requires a policy for handling messages that take longer than the lag constant to arrive. Conversely, local lag is simple to implement using timelines. Consider again “Alice” and “Bob” from our previous example for dead reckoning. As before, Alice’s position is stored in the timeline \textit{alicePos}. In this example, however, Alice’s client uses local lag to set positions in response to her movement commands. We will assume that the local lag constant
Chapter 7. Janus Toolkit and the Timelines Model

is \( \text{DELAY} \). That is, if Alice presses a key to move her avatar, there will be a delay of \( \text{DELAY} \) ms before she observes the movement associated with that key press. Bob should also observe the same movement \( \text{DELAY} \) ms after Alice pressed the key.

Thus, if Alice moves to a new position \( \langle x, y \rangle \), the operation on Alice’s client to process the movement is simply:

\[
alicePos(\text{DELAY}) = (x, y);
\]

That is, the position \( \langle x, y \rangle \) is stored in the \( \text{alicePos} \) timeline \( \text{DELAY} \) ms in the future. For example, if \( \text{DELAY} = 100 \), then Alice’s position is set to \( \langle x, y \rangle \) 100 ms in the future.

As before, Alice’s position is drawn on both clients as:

\[
\text{DrawAvatar}( \text{alicePos}(0) );
\]

That is, the avatar is drawn at its position at the current time.

This very simple code has a range of interesting effects. On Bob’s client, messages indicating Alice’s movements are automatically inserted into the timeline when they arrive. If the message took less than \( \text{DELAY} \) to arrive (hopefully the normal case), then on Bob’s remote client, the new position is inserted into the \( \text{alicePos} \) timeline in the future. For example, if \( \text{DELAY} = 100 \) and the message took 60 ms to arrive, the message is inserted into the \( \text{alicePos} \) timeline on Bob’s computer at \( t = 40 \). This allows the present position of Alice’s avatar to be interpolated (using previously recorded positions). Therefore, the local lag functionality supports smooth movements on remote clients without annoying corrections.

Alternatively, if the message took more than \( \text{DELAY} \) to arrive, say 130 ms, then the positional update is inserted into the \( \text{alicePos} \) timeline in the past \( t = -30 \), and current positions are extrapolated from this (and possibly other) past values.

This simple example illustrates the power of the timelines model. Merely changing the time at which Alice sets her position in the timeline allows a game developer to switch from dead reckoning to local lag. Also, the problems of synchronizing lag between different clients and of dealing with messages which take longer than the lag constant to arrive are handled automatically, requiring no code from the game programmer.
Chapter 7. Janus Toolkit and the Timelines Model

7.2.3 Local Perception Filters

Local perception filters [87] are an example of a consistency maintenance algorithm based on time offsets where remote entities are delayed in time. With the remote lag algorithm, a fixed delay is applied to all remote entities. However, with local perception filters, the key idea is to continually adjust the amount of delay applied to non-player controlled entities depending upon their position relative to the local player’s avatar. We are aware of no games using this technique. This is possibly due to the difficulty of implementing it using existing techniques and toolkits, making it difficult for developers to quickly evaluate how well the approach works in their game. As we shall show, the timelines model makes the implementation of local perception filters tractable, opening its application in distributed game physics.

We first review the algorithm, then show how it can be implemented using timelines. We begin by looking at a single object whose motion is determined by a physics engine, for example in the simple soccer game as shown in Figure 7.3. Through this example we highlight some of the shortcomings of other more commonly used consistency maintenance techniques. In the soccer game, two players kick a ball about in a 2-D world. The players are free to move their avatars around the world, and a physics simulation is used to determine the position of the ball. Ideally, both players would see the ball in the same position, and the ball would react instantly when it is kicked. However, due to network latency, this is not possible. Delayed input techniques such as local lag would provide the desired consistent view. But the local player experiences reduced responsiveness as a delay occurs between kicking the ball and seeing it move. Predictive techniques provide fast response times, but extrapolated positions are often inaccurate, particularly when the avatar changes direction or the ball is kicked. In the presence of physics, such inaccuracies can be highly visible.

Remote lag applies a delay to the remote player’s avatar and renders the local avatar in real-time. This provides immediate responsiveness for the local player. Also, since the position of the remote avatar is time-delayed, interpolation, rather than extrapolation, is used providing smoother animation.

Because the ball travels between the two players’ avatars, when the remote avatar kicks the ball,
the ball’s motion must be delayed by the same amount of time as the remote avatar. Otherwise, the interactions between the player and the ball would appear unrealistic. However, when the ball is near the local player’s avatar, it must move in real-time, or again the interactions would appear unrealistic. Local perception filters balance this by adjusting the ball’s delay depending upon its position relative to the two avatars. For example, if the remote player’s avatar is delayed by 100 ms, the ball is also delayed by 100 ms when it is next to the remote player’s avatar, by 50 ms when it is halfway between the two avatars and by 0 ms when it is near the local player’s avatar. This provides the local player with fast response times as he interacts with the ball immediately. It also allows him to view realistic interactions between the ball and the remote avatar.

To implement this example, a developer therefore needs to continually adjust the delay associated with the ball as it moves relative to the positions of the avatars. As an additional complication, the physics simulation for the ball must be carried out on the client whose avatar is closest to the ball. The delay in rendering the ball on the other client provides sufficient time for the ball’s position to be transmitted over the network. In sum, using local perception filters to implement this simple game requires the programmer to maintain different timeframes for both players, to adjust these timeframes dynamically, and to implement a distributed physics simulation with dynamic
change of simulation host.

Timelines make this implementation tractable because they allow the programmer to easily access the state of an object at any point in time. Specifically, we use timelines to:

• pass the right to calculate the position of the ball using the physics simulation;
• determine the time delay for the ball; and
• determine the time delay for the other player’s avatar.

In our example, we assume a client-server architecture with two clients each running a physics simulation to determine the position of the ball. The server uses the ball and player position information to arbitrate which player is controlling the updates to the ball position. To illustrate how timelines can be used to implement local perception filters, we require three timelines: player1Pos, player2Pos and ballPos representing the positions of the three game entities. We also require one additional timeline, ballControl, that indicates which player is controlling the physics simulation for the ball and updating the ballPos timeline.

The server receives updates for the player and ball positions from each client and then, based on the distances of each player from the ball, determines which player should be updating the ball position. The server then sets this information in the ballControl timeline.

ballControl(0) = Player1;

The following steps then occur on each player’s computer. Here we describe the steps assuming that player 1 is the local player. Then, on player 1’s computer, the client checks the value in the ballControl timeline. If player 1 is controlling the ball, his client calculates the new ball position and sets that value in the ballPos time-line. If player 2 is controlling the ball, player 1’s client does nothing as it will be able to get values from the timeline that are set by player 2’s client. Thus, assuming getUpdatedBallPos is a method that updates the ball position and then returns that updated position, the code for adding new values to the timeline is:

```java
if (ballControl(0) == myPlayerNumber)
{
    ballPos(0) = getUpdatedBallPos();
}
```

Next, to draw the ball in the correct position, the players must determine the delay to apply to the
The ball. Assuming again that player 1 is the local player, then player 2’s avatar is drawn at \( DELAY \) ms in the past. His avatar’s position will be:

\[
\text{player2Pos}(\text{-DELAY});
\]

and the local player’s avatar (player 1) is drawn at:

\[
\text{player1Pos}(0);
\]

To estimate the ball’s position, we use the position where it was last drawn as a first approximation. We will assume this value has been stored in a variable called \( \text{prevBall} \). Then, assuming that the lag applied to the ball is linear depending upon the distance between it and the two players’ avatars, we can calculate the \( \text{ballDelay} \) as follows:

\[
d1 = \text{length}(\text{player1Pos}(0) - \text{prevBall});
\]
\[
d2 = \text{length}(\text{player2Pos}(\text{-DELAY}) - \text{prevBall});
\]
\[
\text{ballDelay} = \frac{\text{DELAY} \times d1}{d1 + d2};
\]

Then the ball position to be rendered is:

\[
\text{ballPos}(\text{-ballDelay});
\]

This example illustrates how timelines’ ability to access entity positions at any point in time has allowed us to easily change the timeframe of the ball each time it is rendered. It also shows how it is possible to allow multiple clients to coordinate updates on shared state data. Given the ease with which local perception filters can be implemented with timelines, it becomes practical for developers to assess their suitability for games they are creating.

### 7.2.4 Frame of Reference for Game-Critical Decisions

As we saw in the Eliminate game in our study in Chapter 5, player experience can sometimes be improved if the outcome of game-critical events are determined based on the view of one of the clients. This is especially true when one of the clients has a greater stake in the outcome of the event, as in games that involve a shooter and a target. However, implementing this can be complex as it requires the server to be capable of unwinding time in order to determine what a client saw at the time of the game-critical event. For example, in a first person shooter, the server must
determine the position of the target avatar on the shooter’s client at the time the shot was fired. Since lag is applied to remote avatars, each client’s frame of reference is different.

We now show how Timelines can be used to rewind time and implement game-critical decisions from different perspectives. Figure 7.4 shows how timelines are used to solve this targeting problem. First we look at how timelines are used to easily render a current version of the local avatar and a time delayed version of the remote avatar. We assume that we have two timelines $\text{avatar}_1$ and $\text{avatar}_2$. These timelines will contain the position of the avatar, the direction it is aiming and an indication of whether or not the avatar is shooting. Without loss of generality, we assume that $\text{avatar}_1$ is currently shooting at $\text{avatar}_2$. These two timelines are shared by two clients and a central server. When players perform input actions, the appropriate timelines are updated at time 0. Thus, if player 1’s avatar has moved to position $(x, y)$, is aiming in direction $(\text{aim}_x, \text{aim}_y)$ and is shooting, then the avatar state is set as follows:

$$\text{avatar}_1(0) = (x, y, \text{aim}_x, \text{aim}_y, \text{TRUE});$$

When rendering the avatars, we want to display the local avatar at the current time, but display a delayed version of the remote player’s avatar. Thus, assuming that the amount of delay to be applied to the remote avatar is $\text{REMOTE}_\text{LAG}$, then, on player 1’s client, the avatars are drawn
Chapter 7. Janus Toolkit and the Timelines Model

as:

\[
\text{DrawAvatar (avatar1 (0));} \\
\text{DrawAvatar (avatar2 (-REMOTE_LAG));}
\]

Now we will look at how timelines can be used to allow the server to determine whether or not player 1 was aiming at player 2 when he fired. First, we need to know the length of time required for a message to travel from the client to the server. If we assume that a message takes $L$ ms to arrive at the server, then, the server must make the hit decision based on the state of the shooter (avatar1) at time $-L$ (when the message was sent) versus the state of the target (avatar2) at time $-L - REMOTE\_LAG$ (where the shooter believed the target was at time $-L$).

Periodically, the server checks the avatar1 timeline to see whether a shot has been fired. The time of the last known shooter status is queried and saved to $t$. The state of the shooter at that time is retrieved, and used to determine whether the shooter was firing his weapon at that time:

\[
t = \text{LastKnownTime(avatar1)}; \\
\text{shooter = avatar1(t);} \\
\text{if(shooter.IsShooting ) ...}
\]

If the player was shooting, the server then determines the state of the target player at the time of firing, as viewed by the shooter. This is done by subtracting the amount of remote lag from the time the shot was fired, and retrieving the target state at that time:

\[
\text{target = avatar2(t - REMOTE\_LAG);} \\
\]

Finally a TargetHit function uses the avatars’ positions and shooter’s direction to determine whether the target avatar was hit. If the target was hit, its health points are decremented.

\[
\text{if (TargetHit (shooter.Position, shooter.Heading, target.Position))} \\
\text{\{ } \\
\text{\quad targetHealth(0) = targetHealth(0) - 1; } \\
\text{\}}
\]

Thus, the server needs to take account of the shooter’s frame of reference when a shot was made to determine whether the shot hit the target. The timelines model’s ease of accessing past states makes this straightforward.
Chapter 7. Janus Toolkit and the Timelines Model

Figure 7.5: Corrections to incorrect states predicted by dead reckoning may be corrected by a) immediately warping to the correct position, b) following a straight path to a future point predicted by dead reckoning, or c) following a curved path.

7.2.5 Correction Techniques

As we saw in Chapter 6, corrections are significantly less noticeable and significantly less annoying when they are performed smoothly over time. However, programming smooth corrections can be challenging. With dead reckoning, when a new positional update arrives that is significantly different from the current predicted location, the simplest solution is to immediately move the entity to the new location, as shown in Figure 7.5(a). But, as our studies in Chapters 5 and 6 showed, this results in jerky animation and can be visually jarring for the player.

With smooth correction techniques, an entity is progressively moved to its correct position. For example, one might aim to have the entity in the correct position 1,000 ms in the future. It is not sufficient to move the entity to its current position 1,000 ms in the future; instead, one must
estimate where it will be in 1,000 ms, and progressively move the entity to that location.

The first step in the algorithm is to select the location and time of the entity’s corrected position (e.g., the new location 1,000 ms in the future.) The entity then moves at increased speed until it reaches this correct location, as shown in Figure 7.5(b). This corresponds to the smooth corrections with constant velocity approach that was described in Chapter 6. Alternatively, the correction may be done using a gradually changing velocity, causing the entity to follow a curved path, as shown in Figure 7.5. This progressive correction can be difficult to program, as one needs to simultaneously deal with entity’s current position, its correct (but stale) position, and its future correct position. Furthermore, the avatar’s position must be updated over time until the correct position is attained.

Smooth positional corrections can be easily specified using timelines. This is accomplished by overriding the position timeline’s default remote update function of Figure 7.2. Normally, remote updates are handled by adding the incoming value to the timeline at the appropriate time. This approach replicates the timeline on all clients that have access to it. For smooth corrections, however, we purposely wish the timelines to diverge – when a local client receives a correction, the local timeline is modified to gradually move toward a consistent state.

Figure 7.6 shows this approach. As shown in Figure 7.6a, we use a timeline to represent the positions of a remote player’s avatar. The current position of the avatar has been extrapolated, based on these known positions. We identify this extrapolated location as `currentPos`. Then, as seen in Figure 7.6b, the client receives a message indicating that the avatar was actually at a position `fixupPos` at some earlier time `fixupTime`. Extrapolating from this position and time, we deduce that the avatar should in fact currently be at position `correctCurrentPos`.

The simple solution to this error is to update the current position to `correctCurrentPos`, and this is in fact what the timelines model does by default.

Instead, we decide to smoothly move the avatar to the correct position over the next 1,000 ms. The avatar’s target position (`targetPos`) is determined by extrapolating 1,000 ms into the future (Figure 7.6c). The avatar will move quickly over the next second to that position (Figure 7.6d).

We will now describe in more detail, how this is accomplished using the timelines model. First, we need to save the current position before the update is applied, i.e., the avatar position at time 0. (This is necessary because inserting a new past value into the timeline will alter the current
Chapter 7. Janus Toolkit and the Timelines Model

Figure 7.6: A timelines implementation of smooth corrections. (a) initial state of the timeline (b) an update arrives which would result in a sudden change in position (c) the update is used to determine a target position for the smooth correction (d) the current position and target positions are inserted into the timeline.

position).

\[ \text{currentPos} = \text{avatarPos}(0); \]

Next, we place the fixup position (the position contained in the latest update message) into the timeline at the correct time. This allows us to estimate a target position we want to arrive at 1,000 ms in the future (Figure 7.6c).

\[ \text{avatarPos(\text{fixupTime})} = \text{fixupPos}; \]
\[ \text{targetPos} = \text{avatarPos}(1000); \]

Finally, we insert both the current position, and the target position into the timeline (Figure 7.6d). (The assignment of the target position is necessary as its extrapolated value changes following the update of the current position.)
avatarPos(0) = currentPos;
avatarPos(1000) = targetPos;

The correction may be either a straight path toward the target position as shown in Figure 7.5(b), or if a nonlinear interpolation function is used, the correction will follow a curved path as in Figure 7.5(c).

This example illustrates how smooth corrections can be easily implemented using timelines. By default, the timeline remote update function inserts values from remote clients into the local copy of the timeline. This function would contain just one step:

avatarPos(fixupTime) = fixupPos;

With the simple addition of four lines of code, the remote update function becomes:

currentPos = avatarPos(0);
avatarPos(fixupTime) = fixupPos;
targetPos = avatarPos(1000);
avatarPos(0) = currentPos;
avatarPos(1000) = targetPos;

and we have implemented smooth corrections. Also, depending upon the form of interpolation and extrapolation functions used we can switch between performing the correction with either a constant velocity or with a velocity that changes more gradually during the correction. The use of a linear interpolation function will cause a step change in the velocity of the entity at the start of the correction and the correction will follow a straight line path as shown in Figure 7.5(b), while the use of a higher order interpolation function will cause the velocity to change more gradually and the correction will follow a smoother path as shown in Figure 7.5(c).

The key concept illustrated in this example is that by having the ability to modify a timeline, programmers are able to explicitly control the divergence of timelines between different clients. The remote client, where the avatar is being controlled, sees neither the error nor the correction.

7.2.6 Using Timelines: Summing Up

In this section, we have shown that timelines can be used to express a wide range of consistency maintenance algorithms including dead reckoning, local lag and local perception filters, as well as
Chapter 7. Janus Toolkit and the Timelines Model

techniques for smooth corrections and for making game-critical decisions from different perspectives.
We have shown that complex algorithms can be expressed with very little code, making it tractable
for developers to experiment with complex and novel consistency maintenance schemes.

7.3 Implementing Timelines

The previous section showed how timelines’ explicit treatment of time provides the necessary infra-
structure for implementing many consistency maintenance techniques used in networked games. We
now discuss how timelines are implemented within the Janus toolkit. The toolkit is named after
Janus, the Roman god of gates, doorways, beginnings and endings. Janus also had the ability to
see into both the past and the future, just as users of the Janus toolkit are able to access previous
and future versions of the game’s state.

The Janus toolkit is written in C# and is compatible with the Unity game development engine
or with any .NET language. It is built on top of the Lidgren Networking Library [63]. Lidgren is a
networking library for the .NET framework which provides a simple API for connecting a client to
a server, and for reading and sending messages. It provides a range of delivery methods including
unreliable, reliable unordered and reliable ordered UDP messaging. The Janus toolkit is currently

7.3.1 Object Model

Within Janus, timelines are implemented as objects descended from the Timeline class. Each
timeline has a base type (the type of the timeline state) and methods implementing interpolation,
extrapolation and remote update handling. The default values of these methods can be overrid-
den to create arbitrary timeline types. Timeline objects also provide Get and Set methods for
retrieving/modifying the timeline’s state.

Each timeline object has a string identifier. If two clients create timelines with the same
identifier, those timelines are automatically synchronized. As shown in Figure 7.2, whenever a new
value is added to a timeline, an update is sent over the network to its remote peers. When the
remote update is received, it is applied to the timeline via its remote update handling function. By
default, this function simply inserts the new state into the timeline at the correct time; as we have seen, overriding this function can allow easy programming of interesting behaviours, such as the smooth corrections of Section 7.2.5.

Known timeline states (i.e., those that have been inserted into the timeline with the \texttt{Set} method) are simply organized into a doubly-linked list, where each state is tagged by its time. The shared state object may be as simple as a single integer, or it may be any arbitrarily complex object containing multiple properties. We have created a variety of standard timeline objects including an integer, a floating point number, 2-D and 3-D position vectors and more complex objects combining position, heading and velocity. Users of the toolkit can either use these existing timeline objects or create a new type of timeline object.

Although the time values stored in the linked list are real times (measured in milliseconds since the epoch), the programmer always accesses states using relative time, where zero (0) means now, +10 represents 10 ms in the future, and -10 represents 10 ms in the past.

### 7.3.2 Interpolation and Extrapolation

A timeline’s \texttt{Get} method is used to retrieve its state at a given time (past, present or future). The \texttt{Get} method uses the timeline’s interpolation and extrapolation functions as necessary to provide values at times when none is known. Default implementations of linear and stepwise interpolation/extrapolation are provided, and developers can create additional functions to provide domain-specific behaviours. For example, with dead reckoning, the extrapolation function may be either first order (based on position and velocity) or second order (based on position, velocity and change in either speed or direction). The function may also be based upon only the most recent update, or it may be based upon two or more previous states. Our timelines implementation supports the use of a variety of extrapolation (and interpolation) functions, and thus all these options are possible merely by selecting a different extrapolation function. Which form of extrapolation function is most suitable depends upon both the type of game and the type of motion [75]. With timelines, the choice of function can be modified at runtime, thus facilitating adaptive techniques such as the use of \textit{position based history} [90] where the extrapolation function changes based on the motion of the entity.
Chapter 7. Janus Toolkit and the Timelines Model

![Diagram of a centralized message router passing messages between Janus peers]

**Figure 7.7:** A centralized message router passes messages between Janus peers

### 7.3.3 Distribution and Networking

From the developer’s point of view, *Janus* has a peer-to-peer architecture. That is, updates are automatically broadcast to all peers that share the same timeline, and all data is fully replicated.

In the current implementation of *Janus* (see Figure 7.7), we have developed a centralized message router to implement this peer-to-peer communication. The message router may be run as a stand-alone application, or it may be run as an additional process as part of one of the client applications. The router is based on a distributed publish and subscribe architecture [39]. As indicated previously, a string identifier is associated with each instance of a timeline object. When a client creates a timeline object with a given identifier, the identifier is passed to the message router and the client automatically subscribes to updates for that object. When a client stores a new value in a timeline object using its Set method, the value and the time associated with it are sent to the message router which forwards the data to all other clients who have subscribed to that timeline object.

By default, the *Janus* Toolkit does little to minimize the number and size of messages passed between clients. However, several features are available that dramatically reduce bandwidth requirements of applications using the toolkit. First, the programmer can set a minimum time interval between updates. Thus, not all changes to the local timeline are propagated over the network. For example if the local client updates positions every 20 ms and the minimum time interval for sending updates is set to 60 ms, then only one third of the updates are sent. Second, each client continually keeps track of which values have actually been sent over the network. Then, prior to
Chapter 7. Janus Toolkit and the Timelines Model

sending an update, the client performs a check to determine whether or not remote clients can accurately predict the new updated state. Only if the remote client is unable to predict the new state within a set error threshold will the new state be transmitted. By setting the size of the error threshold, the programmer can control the fraction of messages that are sent. Auto-adaptive dead reckoning schemes [13] can also be implemented by adjusting the error threshold depending upon the game situation or factors such as network congestion and bandwidth availability.

Finally, the programmer is able to optimize the size and format of messages. By default, Janus uses object serialization to convert objects to a byte array for transmission over the network. This has the advantage of making it simple to create new timeline classes without the need to worry about how the data is transmitted. However, the built in object serialization can generate unnecessarily large messages. Programmers can override the default serialization methods, and are thus able to optimally format the messages passed.

To support synchronization, Janus uses a global clock. Determining and maintaining a global clock on all clients can be a daunting task due to clock drift, network latency and jitter. We have implemented our global clock using the Berkeley algorithm [51]. In our implementation, the message router periodically sends timing messages to all clients. The message router analyzes the timing messages returned from the client, discarding any outliers and then sends updates to the global time back to the client.

7.3.4 Alternative Architectures

The timeline model does not impose any specific architecture on the game. In the Janus toolkit, we chose to implement a message broadcasting underlay with a peer-to-peer overlay. The decision to use a central message router was made purely for simplicity of implementation, and could be replaced by true peer-to-peer message broadcasting techniques. This implementation easily supports a variety of overlay architectures. As shown in figure 7.8, the message communication architecture is determined by the topology of clients’ subscriptions to timeline objects. Figure 7.8(a) shows a peer-to-peer overlay where each client (C1 and C2) subscribes to updates for all timelines (TL1 and TL2). In the client-server model shown in Figure 7.8(b), each client shares a timeline only with the server. That is, client C1 updates timeline TL1 and the message router only propagates
Chapter 7. Janus Toolkit and the Timelines Model

Figure 7.8: The Janus toolkit allows for a variety of overlay architectures including peer-to-peer, client-server and a hybrid model depending upon the subscription list for each timeline

those updates to the server S1. Similarly, client C2 updates timeline TL2 and the message router only propagates those updates to the server S1. Timelines TL3 and TL4 are both updated by the server and the updates for these timelines are forwarded to both clients. In the hybrid model in Figure 7.8(c), timelines TL1 and TL2 are shared only between one client and the server, while timelines TL3 and TL4 are shared between the two clients and timeline TL5 is shared by all.

7.4 Experience

Despite its status as a research prototype, the Janus toolkit has been used by the authors and other developers to experiment with a variety of lag compensation algorithms and to create several multiplayer games based on either Microsoft’s XNA game development library or on the Unity game engine [97]. The games include the Balloon Burst, Truck Pull and Pedal Race exergames [93], the Eliminate 3-D first person shooter, the Speed Demons racing game [86], the Growl Patrol ubiquitous game [61], and the Liber Persistent world game [54, 103, 102]. The toolkit has also been used to implement the OrMiS tabletop military simulation tool [11] and the game orchestration tool, Tabula Rasa [49].

These games were developed by twelve developers, none of whom were authors of Janus itself.
Chapter 7. Janus Toolkit and the Timelines Model

All were students, ranging from undergraduate to Ph.D. level, and most had only passing experience with distributed systems programming.

Although we have largely focused on the development of multiplayer games, timelines can readily be applied to distributed simulation and to groupware applications such as shared editors and drawing tools or chat applications [82].

Our experience indicates that timelines can make it easy to implement basic networking in multiplayer games. As we have seen, timelines also make it tractable to implement sophisticated algorithms. For example, we have experimented with the combination of the local perception filters [87] algorithm with smooth corrections [91] to provide a novel solution to the distributed physics problem. Thus far, we have created a credible simulation involving two players interacting with tens of objects over networks with up to 100 ms of latency.

7.5 Strengths and Limitations

The power of the timelines model lies in its explicit treatment of time. Automatic interpolation and extrapolation mechanisms allow programmers to easily access shared state data from any point in the past or future. This technique is powerful for manipulating shared data, although it is not without limitations. In our example of implementing local perception filters, we have shown how two clients can coordinate updating a single timeline representing the ball position. In general, however, the current timeline implementation does not support multiple clients updating the same timeline, as the updates from one client by default can overwrite updates made by the other client. Overriding the default remote update function can solve this synchronization issue; however, to-date this has been left to the developer using the toolkit to provide the implementation. In future versions of the Janus toolkit, we will provide a variety of options for remote update handling that will support synchronization techniques such as time warp [66], optimistic synchronization protocols [41], conflict merging and/or operational transform [36].

By default, timelines require the entire shared object to be sent over the network for each update. This makes them unsuitable for large data structures. We have begun to explore how timelines can be made more efficient by sending only changes to the shared state, as opposed to
Chapter 7. Janus Toolkit and the Timelines Model

sending the entire object. Using customized serialization methods we are able to transmit only the portions of the data structure that have changed. Further work is required to generalize this solution for all timelines objects.

We have shown how the timelines model makes it easy to access shared state data at any point in time. However, the same is not true for command type data, such as “shoot or “crouch. There is no method to interpolate or extrapolate these types of actions and thus each command must be accessed individually. We have experimented with various options for integrating commands into our timelines model. Some options include, allowing the programmer to access a list of commands that occurred over a time range, or using an event-driven model for commands, possibly delaying events based on the command timestamp.

Our implementation of the global clock has been used successfully to synchronize clients on a variety of computers. We have found that over the local area network we are able to synchronize the clocks generally within a few milliseconds. Testing over wide area networks indicates that clock synchronization within the small tens of milliseconds is achievable. A more sophisticated algorithm may be required under conditions involving higher network latency and jitter. Also, we have not yet implemented safeguards to ensure that changes to the clock occur gradually and that the clock cannot move backward. We have not seen a need for such safeguards in our studies to-date, but this may require consideration in the future.

The amount of memory used by our implementation represents an area for future optimization. Currently all values that are set are stored in the timeline, possibly requiring large memory. Janus currently truncates history to limit storage requirements. We plan to adapt algorithms for compacting history developed for solving the groupware latecomer problem [25], and mechanisms for compressing messages [52].

Timelines provide a novel programming model. For programmers familiar with message passing techniques, the shift to thinking about a shared state model indexable by time can be significant, perhaps analogous to the shift from procedural to object oriented programming. We have found that developers who dive into the model without carefully studying its documentation and examples make the mistake of trying to treat it like a message passing system. As with all novel programming models, developers need to adjust to the model’s way of thinking.
7.6 Conclusion

In this chapter, we have presented the *timelines* programming model for consistency maintenance in networked games. Timelines facilitate the implementation of a variety of consistency maintenance techniques by making the treatment of time an integral part of the programming model. Timelines allow programmers to manipulate past and future values and control how state diverges over time for different players.

Timelines have been implemented within the *Janus* toolkit, and were used to implement all examples presented in this chapter, as well as all the networking components of the games created for the user studies in this thesis.
Networking has become a key technology within digital games, as multiplayer interaction has become an almost ubiquitous feature. Games permit people to closely coordinate the activities of small groups, to engage in large combats involving hundreds of people, and to view and react to the actions of opponents in real-time. However, despite the importance of networking in games, the programming of consistency maintenance remains a difficult problem with network latency making it impossible for players to experience both fast response times and a completely consistent view of the game world.

The research presented in this thesis provides guidance and support for implementing consistency maintenance in networked games. This work was informed by a pair of user studies and through the development of metrics and a design space for consistency maintenance. The metrics provide an empirical comparison among the different approaches. The consistency maintenance design space includes two additional concepts – techniques for resolving game-critical decisions in the presence of inconsistency, and techniques for repairing inconsistencies when they occur. The design space was evaluated in two user studies. The first study investigated the effects of different consistency maintenance approaches on player experience and performance. It demonstrated that players value intuitive local behaviour more than global consistency. It also showed that players prefer smooth animation and prefer the results of game-critical decisions to be consistent with their view of the game, even when this results in lower overall consistency. The second study investigated techniques for reducing the negative impact of repairing inconsistencies. We found that the three
most important factors affecting the detection of corrections are the player’s locus of attention, the
smoothness of the correction and the duration of the correction.

All the networked games used in this research were developed using our Janus toolkit. The
toolkit simplifies the implementation of game networking and has allowed for easy experimentation
with different consistency maintenance techniques.

8.1 Contributions

This research has contributed novel ideas and knowledge to the field of networked games. There
are four major contributions:

1. Inconsistency Metrics: We have defined a set of metrics that describe the algorithmic space
   of techniques for reducing inconsistency and provide a means of performing an empirical
   comparison among the different approaches.

2. Design Space for Consistency Maintenance: We have shown that the characteristics of the
techniques for reducing inconsistency alone are not enough to determine the effects on player
experience. We have established a design space for consistency maintenance that includes
two new design concepts – techniques for resolving game-critical decisions in the presence
of inconsistency, and repair technique. Through a user study, we have provided empirical
evidence of the trade-offs embodied in this design space and have tested the effects of some
of the choices presented in the design space.

3. Repairing Inconsistencies: Our user study investigating repair techniques has shown that
corrections are most problematic when they occur within a player’s locus of attention, and
corrections are far more annoying when they impact the game situation. We have also shown
that smoothing corrections over even a 250 ms interval dramatically reduces the frequency
with which corrections are noticed, and that performing corrections with a gradually changing
velocity is better than using constant velocity. Finally, the likelihood with which a player sees
that a correction has occurred depends more on the correction’s velocity than its magnitude.
Chapter 8. Conclusions and Future Work

4. *Game Networking Infrastructure*: The *Janus* toolkit provides a means for implementing networking in computer games. The toolkit also provides the infrastructure for experimentation with consistency maintenance techniques that can be used in networked games.

### 8.2 Limitations

The primary limitations of this work are:

1. It is often difficult to generalize the results of game performance studies because changing small parameters in a game can possibly result in quite different results. For example in the Eliminate game, the results might differ if the game changed any of: the reload speed, the speed at which the avatars move, the avatar size or the number of players. Although the specific effects we observed in our studies may not be generalizable to all games, our studies show that these effects do occur in some situations and game developers should be aware of the possibilities.

2. The *Janus* toolkit has only been used by developers in the EQUIS lab. Although the applications developed in the lab cover a range of game styles, the limits of the toolkit in terms of numbers of concurrent users and the quantity of shared data have not been thoroughly assessed.

The key assumptions of this thesis are:

1. The games used in the user studies represent a sufficient range of networked games to validate the applicability of the consistency design space for networked games in general.

2. The participants in the consistency maintenance study represent typical gamers and non-gamers.

3. The applications developed in the EQUIS lab using the *Janus* toolkit provide a sufficient range of game styles to illustrate the utility and simplicity of the toolkit for implementing networking in computing games.
Chapter 8. Conclusions and Future Work

8.3 Future Work

This investigation into consistency maintenance leads to a series of open questions. Areas requiring further research are presented in the sections below.

8.3.1 Future Studies

The user studies presented in this thesis were limited in terms of game design, network conditions and the number of concurrent players. These factors could be explored in more detail in future studies. Further studies could investigate the interaction effects among the axes such as the relationship between remote lag and corrections and whether the perspective for game-critical decisions affects the choice of correction technique. Additional studies could also explore other game situations such as interactions in real-time strategy games and collisions between game objects.

8.3.2 Other Algorithms for Consistency Maintenance

Our studies have explored the trade-offs between existing consistency maintenance algorithms. Other algorithms and techniques such as local perception filters or dynamically adjusting the amounts of local and remote lag based on either the game situation or on network conditions could be investigated.

8.3.3 Janus Toolkit

The Janus toolkit was initially developed to run under Windows and primarily for games developed using Microsoft’s XNA game development library. Janus is currently being upgraded to run on multiple platforms and to integrate with the Unity game engine. New features will improve the usability and efficiency of the toolkit. The toolkit is currently available open-source. Improved documentation and additional examples will make the toolkit more accessible for researchers and game developers outside of the EQUIS lab.
Chapter 8. Conclusions and Future Work

8.4 Summary

Consistency maintenance techniques have real effects on a player’s quality of experience. Our research has provided an expanded view of consistency maintenance in networked games, including additional two factors – decision-making and error repair – that can help game designers improve player experience. We have performed user studies confirming that it is not sufficient to focus on consistency alone. We saw that consistency algorithms providing the best consistency do not necessarily lead to the best user experience. Players preferred smooth animation and preferred the results of game-critical decisions to be consistent with their view of the game.

Our Janus toolkit provides game developers with the infrastructure for experimentation with and evaluation of different consistency maintenance techniques.
References


References


References


References


References


References


References


References


149
References


References
