Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marcel Köhler, Andreas Pongs

Weiterentwicklung einer Echtzeit-Kommunikationsbibliothek
für handybasierte Multiplayerspiele

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marcel Köhler, Andreas Pongs

Weiterentwicklung einer Echtzeit-Kommunikationsbibliothek für handybasierte Multiplayerspiele

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai v. Luck
Zweitgutachter: Prof. Dr. Olaf Zukunft

Abgegeben am 5. Juli 2007

Marcel Köhler, Andreas Pongs

Thema der Bachelorarbeit

Weiterentwicklung einer Echtzeit-Kommunikationsbibliothek für handybasierte Multiplayer-spiele

Stichworte

Multiplayer Handy Spiele mobile Netze

Kurzzusammenfassung

Thema dieser Arbeit ist die Realisierbarkeit netzwerkbasierter Echtzeitspiele in aktuellen Mobilfunknetzen. Langfristiges Zeit ist dabei die Erweiterung der Multiplayer-Plattform *Exit Games Neutron*® um diesbezüglich geeignete Funktionen. Dazu wurden innerhalb einer Testapplikation verschiedene Latenzausgleichstechniken implementiert und im Hinblick auf ihre Tauglichkeit für verschiedene Spielgenres in Mobilfunknetzen getestet. Die Auswertung zeigte, dass insbesondere die Verbreitung von UMTS und MIDP2.0 die Umsetzung vieler Echtzeit-Genres in Mobilfunknetzen ermöglicht. Am Ende der Arbeit stehen konkrete Vorschläge zur Realisierung der Multiplayer-Komponente verschiedener Spielegenres, sowie ein Entwurf zusätzlicher Komponenten, um die Neutron zur Unterstützung echtzeitorientierter Multiplayerspiele ergänzt werden könnte.

Marcel Köhler, Andreas Pongs

Title of the paper

Further development of a library for distributed multiplayer games in mobile networks

Keywords

Distributed Mobile Multiplayer Games

Abstract

This study deals with the feasibility of distributed realtime multiplayer games in current mobile networks. A longterm goal is to extent the functionality of the mobile multiplayer platform *Exit Games Neutron*®. Therefor we implemented different techniques that are used in modern commercial video games and distributed military simulators to reduce the effects of network latency. To verify their suitability we conducted testruns regarding various game genres and diverse mobile network conditions. We conclude this work with concrete suggestions on adequate techniques for several game genres. Furthermore we present viable extensions for the neutron multiplayer platform.

Inhaltsverzeichnis

1 Einführung **2**

2 Analyse **5**

2.1 Die Entwicklung verteilter virtueller Umgebungen . . . 5
2.2 Echtzeit-Spielegenres . . . 6
2.3 Mobilfunknetzwerk-Technologien . . . 8
2.4 Netzwerkprobleme in verteilten Simulationen . . . 10
2.4.1 Datentransferrate . . . 11
2.4.2 Latenz . . . 13
2.4.3 Jitter . . . 14
2.4.4 Paketverlust . . . 15
2.4.5 Quality-of-Service-Eigenschaften der Mobilfunknetze . . . 15
2.5 Konsistenz in Multiplayerspielen . . . 16
2.5.1 Techniken zur Wahrung von Konsistenz . . . 16
2.5.2 Probleme durch Inkonsistenzen . . . 19
2.5.3 Synchronisationsalgorithmen . . . 21
2.5.4 Zeitsynchronisation . . . 27
2.6 Minimierung der Latenz . . . 31
2.6.1 Wahl des Transportprotokolls . . . 31
2.6.2 Wahl der Netzwerktopologie . . . 33
2.6.2.1 Client/Server - Architektur . . . 33
2.6.2.2 Peer-to-Peer Architektur . . . 35
2.6.3 Minimierung der gefühlten Latenz . . . 36
2.7 Darstellungsmodelle . . . 37
2.7.1 Dead Reckoning . . . 37
2.7.1.1 Dead Reckoning Modelle . . . 38
2.7.1.2 Time Compensation . . . 40
2.7.1.3 Konvergenzalgorithmen . . . 40
2.7.1.4 Wahl des Schwellwerts . . . 42
2.7.1.5 Pre-Reckoning . . . 43

2.7.2 Position History Based Dead Reckoning 44
2.7.3 Interpolation 47
2.8 Repräsentative Beispiele aus der Welt der PC-Spiele 52
2.8.1 First Person Shooter: Quake 52
2.8.2 First Person Shooter: Half-Life / Counter-Strike 57
2.8.3 First Person Shooter: Unreal Tournament 59
2.8.4 Sportspiel: Online Madden NFL Football 60
2.8.5 Echtzeit-Strategie: Age of Empires III 60
2.9 Zusammenfassung der Analyse 64

3 Entwurf und Realisierung **66**
3.1 Neutron-Echtzeit-Plattform 67
3.2 Realtime Simulator 71
3.2.1 Netzwerk-Abstraktionsschicht 72
3.2.2 Steuerungsmodelle 74
3.2.3 Darstellungsmodelle 77
3.2.4 Betriebsmodi und Optionen 80
3.2.5 Simulation der Netzwerkeigenschaften 83
3.3 Auswertung der Techniken 87
3.3.1 Sportspiel 90
3.3.2 Simulationen 97
3.3.3 3D-Shooter 103
3.3.4 Arcade-Action 115
3.3.5 Fazit der Auswertung 124
3.4 Evaluation des Realtime-Simulators 124
3.5 Erweiterung der Neutron-Echtzeit-Plattform 126
3.5.1 Kapselung der Darstellungsmodelle 126
3.5.2 Serverkomponenten 128
3.5.2.1 Kollision von Entitäten 128
3.5.2.2 Interface für Levelstrukturen 129
3.5.2.3 Lebenszyklus der Entitäten 130
3.5.2.4 Cheatsichere Trefferauswertung 131

4 Zusammenfassung und Ausblick **132**

Kapitel 1

Einführung

Als das Mobiltelefon Mitte der 90er Jahre für den westeuropäischen Normalverbraucher erschwinglich wurde, war kaum abzusehen, dass es zehn Jahre später bereits für viele Benutzer den Fotoapparat und die Videokamera ersetzen würde. Das Handy war immer griffbereit, und schien deshalb dazu prädestiniert, für die verschiedensten Aufgaben zweckentfremdet zu werden. So übernahm es nach und nach zusätzliche Funktionen als Kalender, Taschenrechner, Radio oder Navigationssystem, und entwickelte sich nicht zuletzt auch zur tragbaren Spielkonsole. Dank leistungsstarker Prozessoren sind auf heutigen Handys Spiele möglich, die grafisch auch einen Gameboy Advanced in den Schatten stellen. Durch den Vorteil der Konnektivität bietet das Handy als tragbare Spielkonsole zudem viele neue Möglichkeiten. Parallel zur rasanten Verbreitung der Handys explodierte in den 90er Jahren auch die Zahl der Internet-Nutzer, und netzwerkbasierte Multiplayerspiele entwickelten sich nach der Jahrtausendwende mit Spielen wie *Counter-Strike* und *World of Warcraft* vom Nischenmarkt zur Massenbewegung. Onlinespiele gehören heute zu den am stärksten expandierenden Industriesegmenten weltweit [LLP (2006)]. Die Realisierung von Onlinespielen auf mobilen Endgeräten erscheint also als ein naheliegender Schritt.
Diese Arbeit entstand bei der Firma Exit Games, die mit ihrem Produkt *Neutron*® eine weltweit führende Position im Segment spezieller Multiplayer Plattformen übernommen hat. Die Neutron-Plattform beinhaltet eigene Server und fertige Funktionsbibliotheken, um Spieleentwicklern bei der technischen Realisierung von Multiplayer-Funktionalitäten einen möglichst großen Teil des Aufwands abzunehmen. Zentrale Funktion ist dabei der Austausch von Interaktionsdaten wie beispielsweise der Spielzüge in einem Schachspiel. Neutron bietet unter anderem auch Funktionen zur schnellen Einbindung globaler Highscore-Tabellen, dem Austausch von Chat-Nachrichten oder auch dem Zuordnen von Gegenspielern auf Basis ihrer Spielstärke [1].
Die besonderen Eigenschaften mobiler Funknetze bringen verschiedene Probleme beim Realisieren netzwerkbasierter Multiplayerspiele mit sich. Unzureichende Standards im

[1] die Neutron-Komponenten werden in Sektion 3.1 ausführlich vorgestellt

Hinblick auf den Funktionsumfang der Endgeräte und unsichere Quality-of-Service-Eigenschaften der Mobilfunknetze führen beim Datenaustausch über Neutron momentan zu Verzögerungen in einer Größenordnung von mehreren Sekunden. Ein großer Teil der Multiplayer-Spielgenres, wie z.B. Action- oder Rennspiele, ist mit derart hohen Latenzen nicht zufriedenstellend zu realisieren. Diese zeitkritischen Genres werden unter dem Oberbegriff der „Echtzeitspiele" zusammengefasst, und setzen im Gegensatz zu rundenbasierten Spielen (wie z.B. Schach oder Kartenspiele) eine direkte und stetige Interaktion mit den Mitspielern voraus.

Die Weiterentwicklung standardisierter Laufzeitumgebungen sowie die zunehmende Verbreitung moderner Funknetz-Technologien wie UMTS lassen die technischen Hürden zur Realisierung von handybasierten Echtzeitspielen heute mehr und mehr schwinden. Ziel dieser Arbeit ist diesbezüglich die Untersuchung etablierter Latenzausgleichstechniken aus den Bereichen der vernetzen militärischen Simulationen und der Multiplayer-Videospiele im Hinblick auf ihre Tauglichkeit zur Anwendung in mobilen Netzen. Langfristiges Ziel unseres Auftraggebers Exit Games ist die Erweiterung von Neutron um Funktionen, die wiederverwendbare Lösungen zu typischen Problemen bei der Realisierung von Echtzeitspielen bieten. Unser Aufgabenbereich liegt dabei in der Evaluierung konkreter Algorithmen, die zur Lösung netzwerkbedingter Probleme existieren. Der Schwerpunkt liegt dabei auf einzelnen Ende-zu-Ende-Verbindungen, weshalb wir zusätzliche Problematiken wie Serverlast oder Skalierbarkeit ausklammern.

Im ersten Kapitel des analytischen Teils unserer Arbeit betrachten wir die geschichtliche Entwicklung der verteilten virtuellen Umgebungen und der damit verbundenen Technologien. In Abschnitt 2.2 stellen wir dann die verschiedenen Spielegenres vor, die wir im Rahmen unserer Arbeit auf ihre Realisierbarkeit in mobilen Netzen untersuchen wollen. Abschnitt 2.3 stellt aktuelle Mobilfunknetz-Technologien vor und liefert einen Überblick über die prognostizierte Marktentwicklung. In Sektion 2.4 gehen wir auf die Probleme ein, die in Multiplayerspielen durch Quality-of-Service-Defizite des zu Grunde liegenden Netzwerks entstehen können, und befassen uns in 2.6 mit Software-Technologien, die zur Minimierung dieser Probleme eingesetzt werden. Die lokale Darstellung und Positionierung verteilter Objekte ist diesbezüglich ein zentrales Thema, das wir in Abschnitt 2.7 ausführlich behandeln. In der letzten Sektion unseres Analyse-Teils stellen wir repräsentative Beispiele aus dem Bereich der PC-Spiele vor, und erläutern die jeweils verwendeten Multiplayer-Algorithmen

In Kapitel 2 implementieren und evaluieren wir die wichtigsten Softwaretechniken innerhalb einer Testumgebung. Dazu stellen wir in Abschnitt 3.1 zunächst die Exit Games Echtzeit-Architektur als Grundlage unserer Applikation vor. In 3.1 erarbeiten wir einen konkreten Vorschlag zur Integration von Zeitsynchronisation. In Abschnitt 3.2 stellen wir den Aufbau und Funktionsumfang unserer Testapplikation „Realtime Simulator" vor, und befassen uns in 3.3 mit der Auswertung der Testläufe. Eine abschließende Evaluation unserer Applikation erfolgt in Abschnitt 3.4, bevor wir in Sektion 3.5 auf Basis unserer Untersuchungen mögliche Erweiterungen für die Neutron-Echtzeit-Plattform vorstellen. Kapitel 4 fasst unsere Arbeit

zusammen und liefert einen kurzen spekulativen Ausblick auf die zu erwartende Entwicklung der handybasierten Echtzeit-Multiplayerspiele.

Folgende Themen wurden schwerpunktmäßig von Andreas Pongs bearbeitet:
Einleitung (2.1), Netzwerkprobleme (2.4), Konsistenz (2.5), Time Warp (2.5.3), Client/Server (2.6.2.1), Minimierung der gefühlten Latenz (2.6.3), Pre-Reckoning (2.7.1.5), Interpolation (2.7.3), Quake (2.8.1), Online Madden NFL Football (2.8.4), Age of Empires III (2.8.5), Netzwerk-Abstraktionsschicht (3.2.1), Darstellungsmodelle (3.2.3), Simulation der Netzwerkeigenschaften (3.2.5), Sportspiel (3.3.1), Simulation (3.3.2), Zusammenfassung und Ausblick (4)

Folgende Themen wurden schwerpunktmäßig von Marcel Köhler bearbeitet:
Echtzeit-Spielegenres (2.2), Mobilfunknetz-Technologien (2.3), Rendezvous Mechanismus (2.5.3), Zeitsynchronisation (2.5.4), Reliable UPD(2.6.1), Peer-to-Peer (2.6.2.2), Dead Reckoning (2.7.1), Position History Based Dead Reckoning (2.7.2), Half-Life/Counter-Strike (2.8.2), Unreal Tournament (2.8.3), Neutron Echtzeit Plattform (3.1), Steuerungsmodelle (3.2.2), Betriebsmodi und Optionen (3.2.4), Auswertung der Techniken (3.3), 3D-Shooter (3.3.3), Arcade-Action (3.3.4), Fazit der Auswertung (3.3.5)

Folgende Themen entstanden in Gemeinschaftsarbeit:
Zusammenfassung Analyse (2.9), Evaluation des Realtime-Simulators (3.4), Erweiterung der Neutron-Echtzeit-Plattform (3.5)

Eingetragene Warenzeichen

Neutron® ist ein eingetragenes Warenzeichen der Exit Games GmbH.
Sun® und Java™ sind eingetragene Warenzeichen von Sun Microsystems.
BREW™ ist ein eingetragenes Warenzeichen von QUALCOMM Incorporated.
QUAKE®, QUAKE II™, QUAKE III Arena™ und DOOM™ sind eingetragene Warenzeichen von idSoftware.
Unreal® und Unreal® Tournament™ sind eingetragene Warenzeichen von Epic Games.
Half-Life™ und Counter-Strike™ sind eingetragene Warenzeichen der Valve Corporation.
Age of Empires® ist ein eingetragenes Warenzeichen der Microsoft Corporation.
Bomberman™ ist ein eingetragenes Warenzeichen von HUDSON SOFT.
Guild Wars™ ist ein eingetragenes Warenzeichen der NCsoft Corporation.
Warhammer® 40.000: Dawn of War™ sind eingetragene Warenzeichen von Games Workshop Ltd.

Kapitel 2

Analyse

2.1 Die Entwicklung verteilter virtueller Umgebungen

Echtzeitspiele fallen unter den Oberbegriff der *Distributed Virtual Environments* (DVEs). Mit diesem Begriff werden virtuellen Welten bezeichnet, in denen mehrere Akteure zeitgleich über ein Netzwerk interagieren. Bei DVEs steht das Eintauchen in die virtuelle Welt und damit der Echtzeit-Aspekt im Vordergrund. „Echtzeit" bedeutet hier, dass die Zeit in der simulierten Welt möglichst genau im gleichen Tempo voranschreitet, das auch der menschlichen Wahrnehmung entspricht. Im Gegensatz zu analytischen Simulationen sind Ungenauigkeiten innerhalb der Simulation oft tolerierbar, solange sie außerhalb der menschlichen Wahrnehmung liegen.
Die Entwicklung von DVEs ging zeitgleich von zwei verschiedenen Lagern aus, dem Militär und der Computerspiele-Industrie. Anfang der 80er Jahre investierte das US-Militär große Summen in die Entwicklung verteilter Simulationen, die als Trainigsapplikationen für Soldaten dienten. In den Jahren 1983-1989 wurde im Auftrag der US-Behörde DARPA (Defense Advanced Research Projects Agency) das Projekt SIMNET (SIMulator NETworking) realisiert, dessen Ziel die Vernetzung mehrerer autonomer Simulatoren wie z. B. Panzer-Simulatoren war. Das Projekt war ein großer Erfolg, und zog die Entwicklung des *Distributed Interactive Simulations* - Standards (DIS) nach sich, das die Vernetzung aller proprietären Schlachtfeld-Simulationen der Army, Air Force und Navy ermöglichte. DIS beinhaltet Protokolle mit Synchronisationsalgorithmen aus dem Bereich der analytischen Simulationen in verteilten Systemen, und stellt möglicherweise bis heute die komplexeste Technologie im Bereich der DVEs dar.
Als zweites Lager begann die Computerspiele-Industrie Anfang der 90er Jahre mit der Entwicklung kommerzieller verteilter Echtzeit-Simulationen, in denen mehrere Spieler über lokale Netzwerke und später auch über das Internet mit- und gegeneinander spielen konnten. Vorläufer dieser Entwicklung war das textbasierte Netzwerkspiel *MUD* (Multi User Dungeon), das zu Beginn der 80er Jahre an der University of Essex in England entwickelt wurde. In MUD

konnten zum ersten Mal mehrere Spieler von verschiedenen Computern aus gleichzeitig in eine persistente virtuelle Welt einloggen, und gemeinsam Aufgaben lösen oder gegeneinander kämpfen.

2.2 Echtzeit-Spielegenres

Unter den Oberbegriff der Echtzeit-Multiplayerspiele fallen mehrere Genres, die wir im Folgenden vorstellen.

Rollenspiele

In Rollenspielen steht das Eintauchen in eine virtuelle Welt im Vordergrund. Der Spieler erstellt sich dazu zunächst einen Spielcharakter („Avatar“), dessen grundlegende Eigenschaften und Fähigkeiten anhand von Zahlenwerten repräsentiert werden. Die allgemeine Erfahrungsstufe eines Charakters wird als *Level* bezeichnet, der nach der Charaktererstellung zunächst Level 1 beträgt. Während der Spieler die Welt bereist sammelt er *Erfahrungspunkte*, mit denen er für das Erfüllen von Aufträgen („Quests“) und das Besiegen feindlicher Monster belohnt wird. Zu fest vorgegebenen Erfahrungswert-Intervallen steigt der Spielcharakter um einen weiteren Level auf, und kann dabei seine Fähigkeitswerte verbessern und neue Talente erlernen.
Der Reiz von Rollenspielen liegt im Weiterentwickeln des Charakters, dem Sammeln seltener Schätze und besserer Ausrüstungsgegenstände, sowie dem Erkunden der virtuellen Welt. Bei Online-Rollenspielen macht die soziale Komponente einen großen zusätzlichen Reiz aus. Die Spieler helfen sich gegenseitig und spezialisieren sich gezielt auf bestimmte Fähigkeiten, um sich gut zu ergänzen und so als Team erfolgreicher zu sein. In modernen Online-Rollenspiele können normalerweise mehrere tausend Spieler miteinander interagieren. Spiele mit sehr hohen Spieleranzahlen werden durch die zusätzliche Bezeichnung *Massively Multiplayer* von herkömmlichen Multiplayerspielen unterschieden, bei denen die maximal mögliche Spieleranzahl normalerweise auf etwa 32-64 Spieler begrenzt ist. Eine gängige Bezeichnung für Online-Rollenspiele ist der Begriff *MMORPG*, was für *Massively Multiplayer Online Roleplaying Game* steht.

Echtzeit-Anforderungen:
Rollenspiele sind im Hinblick auf die Echtzeit-Anforderungen relativ genügsam. Normalerweise wird nur per Mausklick oder Tastendruck der Befehl zum Angriff eines Gegners gegeben, und der Kampf findet danach automatisch statt. Sieg oder Niederlage werden hauptsächlich auf Basis der Fähigkeitswerte und der Ausrüstung entschieden, nicht durch exaktes Zielen oder schnelle Reaktion. Dementsprechend sind auch Verzögerungen von mehr als einer Sekunde gut zu tolerieren.

Fast alle Online-Rollenspiele verzichten auf eine Kollisionsabfrage. Durch diese Einschränkung verringern sich die Anforderungen an die Netzwerkanbindung enorm. Bei den wenigen Vertretern, die eine Kollisionsabfrage unterstützen (Guild Wars, Neverwinter Nights), findet die Kollision nur in speziellen begrenzten Gebieten statt, in denen die maximale Spieleranzahl auf etwa 32 Spieler limitiert ist.

Prominente Vertreter:
World of Warcraft, Everquest, Guild Wars, Lineage. Sonderfall: Second Life

Echtzeit-Strategiespiele (Realtime Strategy, RTS)

In diesem Genre kommandieren normalerweise 2 bis 8 Spieler eigene Armeen, die oft aus mehreren hundert Spielobjekten bestehen. In den meisten RTS-Spielen errichten die Spieler eigene Basen und Fabriken, um neue Einheiten zu produzieren.

Echtzeit-Anforderungen:
Das RTS-Genre stellt im Hinblick auf die Performanz des Netzwerks einen der genügsamsten Vertreter im Bereich der Echtzeit-Spiele dar. In Abschnitt 2.8.5 stellen wir mit *Age of Emires III* einen prominenten Vertreter dieses Spieltyps vor, und erläutern die eingesetzte Multiplayer-Technologie.

Prominente Vertreter:
Age of Empires, Command & Conquer, Starcraft, Warcraft , Warhammer 40k: Dawn of War

Simulationsspiele

Der Oberbegriff der Simulationsspiele fasst Subgenres wie Flugsimulationen, Weltraum-Shooter und auch Rennspiele zusammen, wobei die letzte Kategorie den mit Abstand größten Marktanteil besitzt. Hier sind auch Spiele mit absichtlich geringem Realismusgrad vertreten, deren Schwerpunkt auf schneller Action, spektakulären Crashs und/oder irrwitzigen Streckenverläufen liegt. In einigen dieser "Fun-Racer“ können gegnerische Fahrer nicht nur durch Aktionen wie Rammen und Abdrängen aus dem Verkehr gezogen werden, sondern auch durch Waffen wie Minen oder Raketenwerfer.

Echtzeit-Anforderungen:
Exakte Kollisionsabfragen bringen hohe Anforderungen an das zu Grunde liegende Netzwerk mit sich. Je schneller das gesteuerte Fahr- oder Flugzeug in Simulationsspielen die Richtung ändern kann, desto niedriger muss die maximal tolerierbare Netzwerkverzögerung („Latenz“) angesetzt werden. In Abschnitt 2.5 beschäftigen wir uns ausführlich mit den Problemen, die zu hohe Latenzen mit sich bringen.

Prominente Vertreter:
Need for Speed, Trackmania, Microsoft Flight Simulator, Wing Commander

Actionspiele

Prominenteste Vertreter dieses Genres sind heutzutage die *First-Person-Shooter* (FPS, „Ego-Shooter") wie z. B. die Serien *Half-Life*, *Counter-Strike* oder *Unreal Tournament*. Der Spieler steuert dabei seine Spielfigur aus der Ich-Perspektive durch eine Spielumgebung („Map"), und versucht, so viele seiner Gegner wie möglich abzuschießen. Schwerpunkt bei Online-Actionspielen ist der reaktionsschnelle Einsatz von Schusswaffen, wobei im Gegensatz zu Rollenspielen exakt gezielt werden muss. Jede Waffe lässt sich einer der zwei folgenden Kategorien zuordnen:

1. Instant-Hit-Waffen
Ein Schuss aus diesen Waffen schlägt sofort nach Drücken des Feuerknopfs im Ziel ein.

2. Projektilwaffen
Beim Abfeuern dieser Waffen werden die Projektile als eigenständige Objekte modelliert, die sichtbar durch die Spielwelt fliegen und mit anderen Objekten oder der Spielumgebung kollidieren können.

Echtzeit-Anforderungen:
Dieses Genre stellt zusammen mit Rennspielen die höchsten Anforderungen an die unterliegende Netzwerktechnik. Kollisionen sind in aktuellen Actionspielen nicht nur zwischen den von Spielern gesteuerten Avatare möglich, sondern müssen auch für die Projektile berücksichtigt werden. Die geringsten Anforderungen an das Netzwerk lassen sich erzielen, falls auf Projektilwaffen und auf Kollisionen verzichtet wird. Die genauen Hintergründe werden in Sektion 2.4 erläutert. Als das netzwerktechnisch anspruchsvollste Genre unter den Echtzeitspielen werden wir in den Abschnitten 2.8.1, 2.8.2 und 2.8.3 die Multiplayer-Technologien der drei wichtigsten Vertreter *Quake*, *Half-Life / Counter-Strike* und *Unreal Tournament* untersuchen.

2.3 Mobilfunknetzwerk-Technologien

Dieser Abschnitt stellt die in Europa aktuell genutzten Mobilfunknetzwerke vor. Im Hinblick auf die technischen Hintergründe sei an dieser Stelle auf die Quellen [J. Gundermann (2004)] und [Schiller (2003)] verwiesen. Für unsere Arbeit sind in erster Linie die technischen Eigenschaften im Hinblick auf die Tauglichkeit für Echtzeitspiele interessant, sowie

die aktuelle und die für die nächsten Jahre prognostizierte Verbreitung.

GSM

Das *Global System for Mobile Communications* (GSM) ist ein Standard für volldigitale Mobilfunknetze, der hauptsächlich für Telefonie, aber auch für leitungsvermittelte und paketvermittelte Datenübertragung sowie für Kurzmitteilungen (Short Messages) genutzt wird. Es ist als Nachfolger der analogen Systeme der erste Standard der so genannten zweiten Generation („2G"), und zur Zeit noch der weltweit am meisten verbreitete Mobilfunk-Standard. Wird ein GSM-Kanal für Datenübertragung genutzt, erhält man eine nutzbare Datenrate von maximal 14,4 kbit/s.

GPRS

GPRS (General Packet Radio Service) ist ein paketorientierter Übertragungsdienst, der auf GSM aufsetzt. Die GPRS-Technik erzielt durch die Bündelung aller GSM-Zeitschlitze eines Kanals in der Praxis eine Datenübertragungsrate von bis zu 55,6 kbit/s.

UMTS

UMTS steht für *Universal Mobile Telecommunications System*, und stellt die dritte Generation („3G") der Mobilfunk-Standards dar. UMTS unterscheidet sich vom Vorgängersystem GSM vor allem durch eine neue Funkzugriffstechnik namens *Wideband CDMA*, die wesentlich höhere Übertragungsraten und schnellere Antwortzeiten ermöglicht.

Abbildung 2.1 zeigt den aktuellen und prognostizierten Anteil der Technologien, die in Westeuropa von mobilen Endgeräten unterstützt werden. GPRS stellt die zur Zeit meist genutzte Technologie zur Datenübertragung in mobilen Netzen dar, während GSM diesbezüglich schon heute mit einem Marktanteil von unter 10% kaum eine Rolle spielt. UMTS ist weltweit auf dem Vormarsch, und wird nach aktuellen Prognosen [Heng (2006)] GPRS bis zum Jahr 2010 von seiner Vormachtsstellung abgelöst haben.

Laufzeitumgebungen für mobile Endgeräte

Mit den Laufzeitumgebungen *Java Microedition* (J2ME) der Firma Sun Microsystems, sowie *Binary Runtime Environment for Wireless* (BREW) existieren zwei Standards zur Entwicklung plattformunabhängiger Applikationen für mobile Endgeräte. BREW ist in Asien und

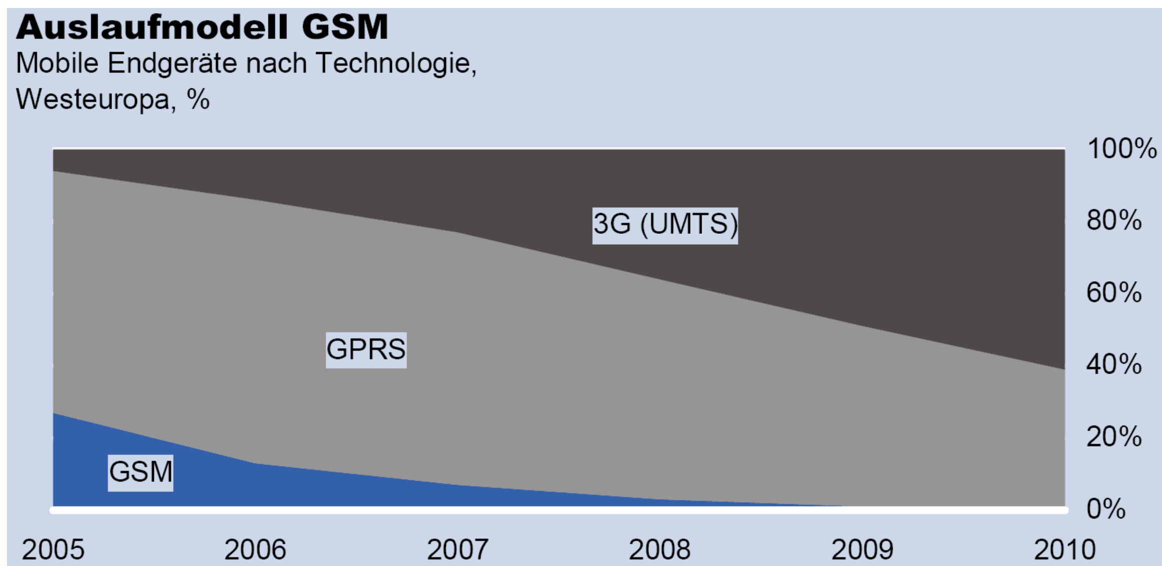


Abbildung 2.1: Anteil der unterstützten Mobilfunknetze in Westeuropa (Quelle: [Heng (2006)])

Nordamerika stark verbreitet, während momentan in Europa fast ausschließlich J2ME unterstützt wird. Innerhalb von J2ME existieren die beiden zusätzlichen Standards *Mobile Interface Device Profile* (MIDP) und *Connected Limited Device Configuration* (CLDC), die den Funktionsumfang der Laufzeitumgebung sowie den Leistungsumfang eines Endgeräts beschreiben, beispielsweise im Hinblick auf den verfügbaren Arbeitsspeicher. Aktuelle Handys unterstützen die Standards MIDP2.0 und CLDC1.1.

2.4 Netzwerkprobleme in verteilten Simulationen

Netzwerkbasierte Multiplayer-Spiele fallen in die Kategorie der ereignisorientieren verteilten Simulationen. Es treten dabei zu bestimmten Zeitpunkten Ereignisse („Events“) ein, die sich auf den Zustand einer oder mehrerer Variablen der simulierten Welt auswirken, und die oft neue Ereignisse auslösen. Alle Ereignisse werden von der Simulationslogik nacheinander in der Reihenfolge ihres Eintrittszeitpunktes abgearbeitet. In unserem Anwendungsgebiet entsprechen die Ereignisse den Interaktionen der Mitspieler. Je nach Spielgenre kann ein Ereignis z. B. das Drücken der Feuertaste darstellen, oder auch komplexere Kommandos beinhalten, wie beispielsweise *erteile Einheit X den Befehl zum Vorrücken auf Position Y*. Die Ereignisse werden in Form von Datenpaketen an die anderen Teilnehmer gesendet. Das Übertragen dieser Ereignisse ist den Einschränkungen des zu Grunde liegenden Netzwerks unterworfen. Dabei sind folgende Faktoren von Bedeutung:

2.4.1 Datentransferrate

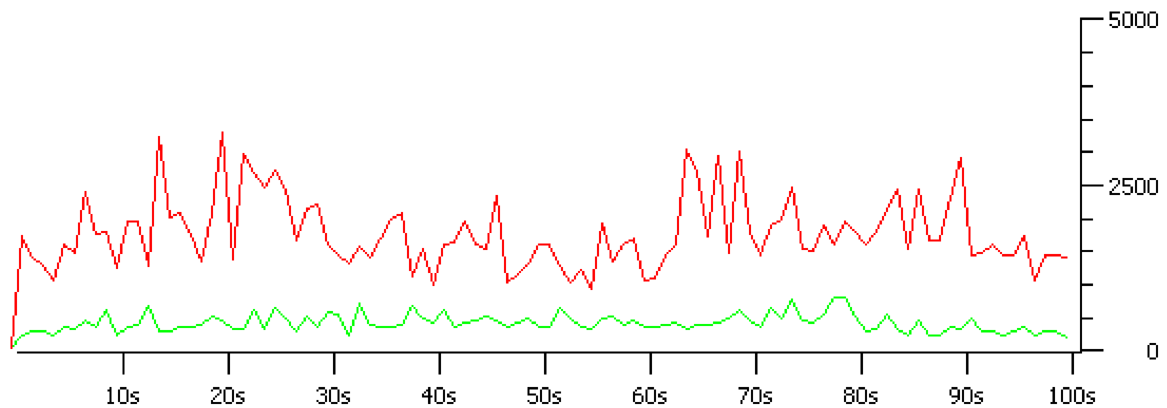


Abbildung 2.2: Netzwerklast MMORPG (Guild Wars) [Bytes/s]

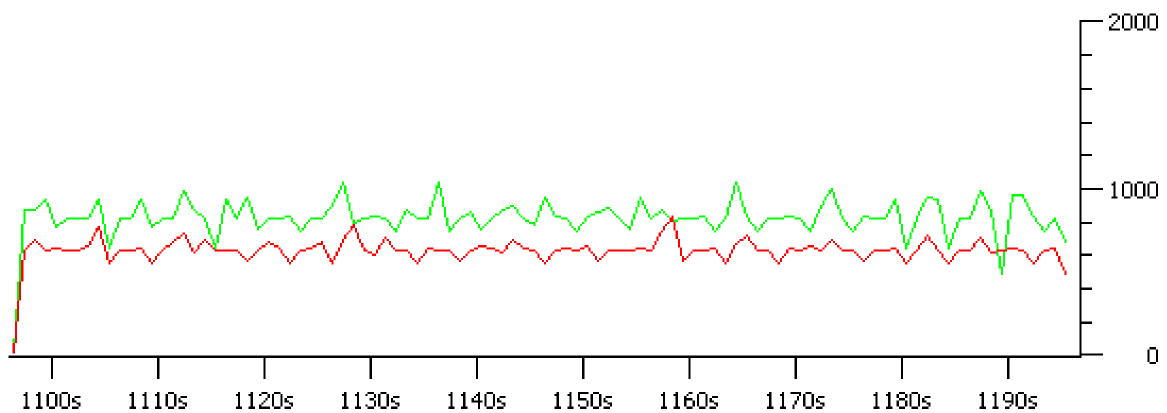


Abbildung 2.3: Netzwerklast Echtzeit-Strategiespiel (Dawn of War), 4 Spieler [Bytes/s]

Die Datentransferrate beschreibt den maximalen Datendurchsatz in Bits pro Sekunde, der zwischen zwei Endgeräten in einem Netzwerk möglich ist. Die begrenzte Datentransferrate schränkt in Echtzeitspielen potentiell die maximale Anzahl an Spieleintitäten und die maximal mögliche Frequenz der Updates ein, in der Aktualisierungen der Spielobjekte übertragen werden. In der Praxis reichen in internetbasierten Echtzeitspielen meist kleine Pakete mit Nutzdaten von wenigen Bytes aus, die aber in einer relativ hohen Frequenz (im Action-Genre etwa 5 Pakete pro Sekunde) übertragen werden.
Wir analysieren im Folgenden die Netzwerklast, die typische Vertreter verschiedener Echtzeitgenres verursachen. Wir benutzten dazu das Netzwerk-Analyse-Tool *Wireshark* [2], um

[2]URL: www.wireshark.org ; Nachfolger von Ethereal

den Datenaustausch während typischer Spielsituationen zu protokollieren und grafisch auszuwerten. In den Abbildungen 2.2 bis 2.5 zeigt jeweils die grüne Kurve die Datentransferrate des ausgehenden, und die rote Kurve die des eingehenden Netzwerkverkehrs an.

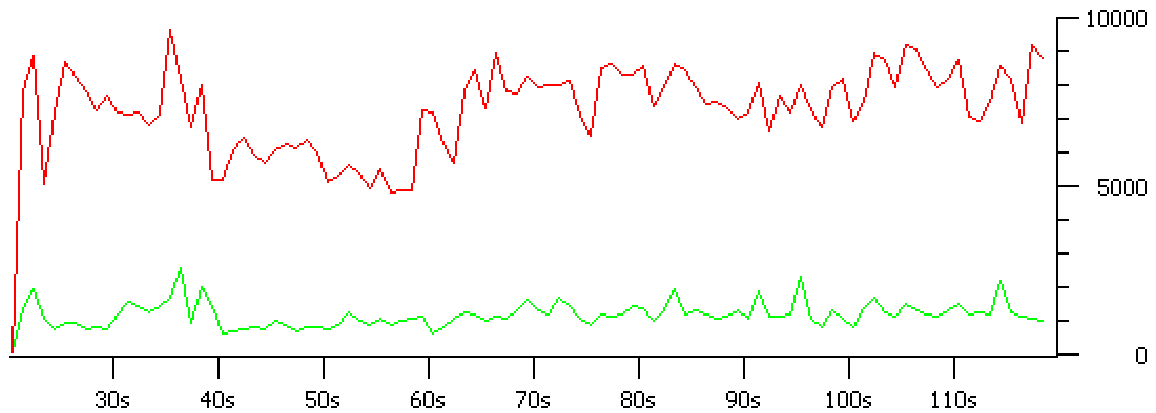


Abbildung 2.4: Netzwerklast Rennspiel (Trackmania Nations), 20 Spieler [Bytes/s]

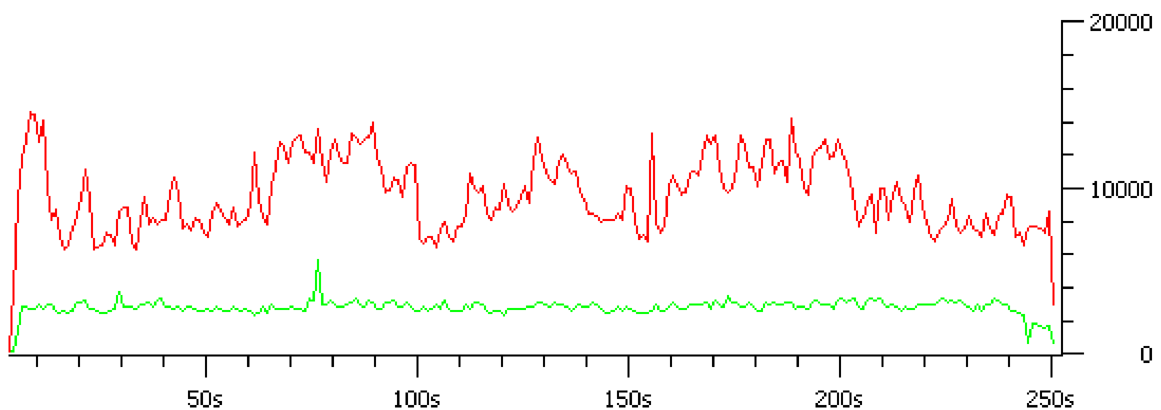


Abbildung 2.5: Netzwerklast 3D-Shooter (Counter-Strike), 16 Spieler [Bytes/s]

Bei den Spielen *Counter-Strike* und *Trackmania* war es jeweils möglich, im Optionsmenü des Spiels die Art der Netzwerkverbindung anzugeben, damit das verursachte Datenaufkommen an die zur Verfügung stehende Bandbreite angepasst werden kann. Bei den Tests gaben wir jeweils eine 54k-Modem-Verbindung an, da dies die Option mit der geringsten Bandbreite darstellte. Für das bloße Auge waren bei dieser Einstellung kaum Einschränkungen im Vergleich zu einer voll genutzten DSL-Verbindung sichtbar. Es ist also davon auszugehen, dass die benötigte Bandbreite bei diesen Spielen noch weiter gesenkt werden kann, und dass eine zu hohe Bandbreite nicht automatisch ein K.O.-Kriterium im Hinblick auf die Realisierbarkeit in einem bestimmten Mobilfunknetz bedeuten muss. Durch das Herabsetzen der maximal

möglichen Spieleranzahl ergeben sich hier weitere Optimierungsmöglichkeiten.
Überraschenderweise war im Echtzeitstrategiespiel *Dawn of War* der ausgehende Datenverkehr eines einzelnen Spielers zum Spieleserver etwas höher als der vom Server empfangene Datenverkehr. Wir hatten ein umgekehrtes Verhältnis erwartet, zumal der Server jedem Spieler die Interaktionen seiner drei Gegenspieler übermitteln muss.

SPIELGENRE	EINGEHEND [KBIT/S]	AUSGEHEND [KBIT/S]
Actionspiel (Counter-Strike), 16 Spieler	88,0 kbit/s	24,0 kbit/s
Rennspiel (Trackmania), 32 Spieler	64,0 kbit/s	12,0 kbit/s
MMORPG (Guild Wars)	16,0 kbit/s	4,8 kbit/s
RTS (Dawn of War), 4 Spieler	4,8 kbit/s	7,2 kbit/s

Tabelle 2.1: durchschnittlicher Datendurchsatz in aktuellen PC-Spielen

2.4.2 Latenz

Als Latenzzeit wird allgemein die zeitliche Verzögerung zwischen einer Aktion und der entsprechenden Reaktion bezeichnet. Im Bereich der verteilten Systeme beschreibt die Latenzzeit die Verzögerung, die durch den Transport eines Datenpakets von einem Endgerät zum einem anderen entsteht. Diese Verzögerung ergibt sich zum einen durch die Ausbreitungsgeschwindigkeit der Daten durch das zugrunde liegende Netzwerk, und zum anderen durch die Verarbeitungsgeschwindigkeit der beteiligten Endgeräte. In der Praxis ist es nicht ohne weiteres möglich, die Verzögerung eines einfachen Verbindungswegs exakt zu ermitteln. Stattdessen wird die *Round Trip Time* (RTT) gemessen. Die RTT bezeichnet die Zeitspanne, die nach dem Versenden eines Pakets bis zum Empfang des Antwortpakets vergeht. Setzt man symmetrische Verzögerungen voraus, kann die Latenzzeit durch RTT / 2 ermittelt werden.

Die Latenz stellt in verteilten Echtzeit-Simulation den kritischsten Parameter dar. Studien belegen, dass der Mensch Verzögerungen wahrnimmt, sobald sie jenseits der 100ms Grenze liegen [Bailey (1982)]. In SIMNET war eine Zielvorgabe, die Latenzen unterhalb der menschlichen Reaktionszeit von ca. 250ms zu halten. Im Nachfolger DIS wurde eine ähnliche zeitliche Obergrenze (300ms) für „vage zusammenhängende Aktionen“ festgelegt. In diese Kategorie fällt beispielsweise das lokale Drücken des Feuerknopfes, und die zugehörige Darstellung des Mündungsfeuers bei allen entfernten Teilnehmern. Eine Grenze von 100ms wurde für „eng zusammenhängende Aktionen“ gesetzt, bei denen feine zeitkritische Interaktionen eine Rolle spielen, wie z. B. beim Formationsflug von Fliegerstaffeln [Standards Committee on Interactive Simulation (SCIS)]

Die Latenz wirkt sich je nach Spielgenre und Implementierung mehr oder weniger stark auf mindestens eins der folgenden drei Spielkriterien aus:

Synchronität

Synchronität bezeichnet die zeitgleiche Darstellung des Spielgeschehens bei allen Mitspielern. Eine exakt synchrone Sicht ist aufgrund der verschiedenen Latenzen nicht möglich, da die Zeitsynchronisation in verteilten Systemen nur bis zu einer gewissen Genauigkeit realisierbar ist (siehe Sektion 2.5.4).

Konsistenz

Konsistenz ist gegeben, wenn alle Teilnehmer die gleiche Sicht auf das Spielgeschehen haben, wenn auch zu leicht unterschiedlichen Zeitpunkten. Aufgrund der Möglichkeit von Paketverlust impliziert das Gewährleisten von Konsistenz die Bestätigung empfangener Nachrichten aller Mitspieler. Das kostet Zeit, und wirkt sich deshalb negativ auf die Interaktivität aus (s. u.).

Interaktivität

Eine hohe Interaktivität bezeichnet eine möglichst verzögerungsfreie Umsetzung der Spieleraktionen auf den Zustand seiner Spielfigur und auf den Zustand der restlichen Spielwelt. Die Interaktivität steht aufgrund der Netzwerklatenzen somit im Konflikt zur Konsistenz. Je actionlastiger das Spielgenre ist, desto weniger akzeptabel ist es, Spielaktionen erst von einem entfernten System bestätigen zu lassen. Verzögerungen beeinträchtigen insbesondere beim Steuern einer Spielfigur das Spielgefühl erheblich. Deshalb ist es bei der Realisierung von Netzwerkspielen in der Regel notwendig, Kompromisse zwischen Konsistenz und Spielfluss einzugehen. Welche Probleme dabei entstehen, und welche Techniken zur Erzwingung von Konsistenz existieren, wird in Sektion 2.5 ausführlich behandelt.

2.4.3 Jitter

Je nach Implementierung einer DVE kann eine stark variierende Übertragungsverzögerung („Jitter“) zu einer Beeinträchtigung des Realismusgrades und des Spielgefühls führen. Welche Implementierungstechniken davon betroffen sind, wird in Kapitel 3.3 genauer untersucht. Jitter kann mit Hilfe eines Eingangspuffers kompensiert werden, was mit einer Erhöhung der durchschnittlichen Latenz erkauft wird. DIS schreibt bei Datenpaketen, die Voice-over-IP Daten transportieren, eine maximale Varianz von 50ms vor.

2.4.4 Paketverlust

Für die meisten Datenpakete innerhalb einer DVE ist normalerweise eine „best-effort“ Übertragung ausreichend. „Best effort“ bedeutet, dass zu Gunsten einer niedrigen Latenz auf eine zuverlässige Übertragung verzichtet wird. Den weitaus größten Teil der Netzwerkkommunikation machen in DVEs periodische Aktualisierungen des Zustands (u. a. Position und Geschwindigkeit) der Spielobjekte aus. Da verloren gegangene Pakete in der Regel veraltet sind, bis sie erneut angefordert und gesendet werden konnten, ist eine erneutes Anfordern dieser Daten überflüssig. Neben den periodischen Aktualisierungen existieren jedoch auch wichtige Ereignisse, deren korrekte Übertragung sichergestellt werden muss. Sie werden dazu im Übertagungsprotokoll als „reliable“ markiert. Zu diesen Ereignissen wird beispielsweise das Abfeuern von Waffen gezählt.

2.4.5 Quality-of-Service-Eigenschaften der Mobilfunknetze

Wir betrachten nun die durchschnittlichen Verzögerungen, die Paketverlustraten und den Datendurchsatz, der in den für uns relevanten Mobilfunknetzen in der Praxis erreicht wird. Dazu haben wir das arithmetische Mittel der wiederum jeweils im Durchschnitt beobachteten Werte aus verschiedenen Studien ([Timm-Giel (2004)], [J.Chesterfield (2005)], [Holma und Toskala (2006)], [Claypool u. a. (2006)]) gebildet. Erste Messwerte aus Exit Games internen Tests lieferten durchgehend etwas bessere Werte. Da diese Messungen auf dem Datenaustausch mit dem in unmittelbarer Nähe stationierten Neutron-Server basieren, betrachten wir diese Werte aber als Best-Case-Szenario. Das Worst-Case-Szenario stellt in mobilfunkbasierten Netzwerkspielen den vollständigen Verbindungsabbruch dar. Mit Abnehmen der Signalstärke verschlechtern sich die QoS-Werte zunehmend, sodass ein angenehmes Spielerlebnis ab einem gewissen Punkt unmöglich wird. Dabei ist außerdem zu beachten, dass der maximal mögliche Datendurchsatz proportional zur Bewegungsgeschwindigkeit des Endgerätes abnimmt. Um ein Echtzeitspiel als tauglich für Mobilfunknetze erklären zu können, sollte zumindest ein weitgehend störungsfreies Spielerlebnis bei der Fahrt in Bussen und S-/U-Bahnen gewährleistet sein.

MOBILFUNKNETZ	DOWNLINK	UPLINK	RTT	PAKETVERLUSTRATE	JITTER
GPRS (mobil)	18,5 kbit/s	6,5 kbit/s	916 ms	< 5 %	15%
GPRS (stationär)	27,9 kbit/s	6,5 kbit/s	916 ms	< 5 %	15%
UMTS	105,0 kbit/s	48,0 kbit/s	340 ms	< 1 %	7 %

Tabelle 2.2: QoS-Eigenschaften aktueller Mobilfunk-Technologien

Die Datentransferrate stellt also in der Praxis nur unter GPRS für Renn- und Actionspiele einen kritischen Faktor dar. In jedem Fall ist es jedoch wünschenswert, den durchschnitt-

lichen Datendurchsatz gering zu halten, da zum Zeitpunkt der Erstellung dieser Arbeit Flatrate-Verträge im Bereich der Mobilfunknetze noch immer nicht stark verbreitet sind. Stattdessen richten sich die Kosten nach der Menge der übertragenen Daten. In wie weit Renn- und Actionspiele möglicherweise trotzdem unter GPRS realisierbar sind, untersuchen wir genauer in Kapitel 3.3.

2.5 Konsistenz in Multiplayerspielen

Wir haben in Abschnitt 2.4 gezeigt, dass bei der Realisierung von Echtzeitspielen ein hoher Grad an Interaktivität mit einem eingeschränkten Grad an Konsistenz erkauft werden muss. Wie negativ sich Inkonsistenzen auf den Spielspaß auswirken, ist je nach Spielgenre sehr unterschiedlich. In rundenbasierten Strategiespielen beispielsweise ist die Position der Einheiten ein zentraler Faktor beim Treffen von Spielentscheidungen, und eine inkonsistente Sicht auf das Spielgeschehen dementsprechend nicht zu tolerieren. In anderen Genres wie z. B. Echtzeit-Rollenspielen können temporäre Abweichungen normalerweise in Kauf genommen werden. Je wichtiger die Konsistenz in einem Spiel ist, desto stärker sind die Einschränkungen, die auf Seiten der Interaktivität gelten, da alle Aktionen erst von der Gegenseite bestätigt werden müssen. Dem Spieleentwickler stehen verschiedene grundlegende Ansätze zur Verfügung, um für ein Spiel einen individuellen Kompromiss zu finden.

2.5.1 Techniken zur Wahrung von Konsistenz

Vollständige Synchronisation

Die stärkste Form von Konsistenz entsteht, wenn eine vollständige Synchronisation des Spielverlaufs bei allen Teilnehmern erzwungen wird. In einer Client/Server-Architektur kann dies durch das *Minimal Client Prinzip* erreicht werden. Beim Minimal Client Prinzip müssen sämtliche Aktionen erst vom Server bestätigt werden, bevor sie lokal ausgeführt und angezeigt werden. In Peer-to-Peer-Architekturen wird perfekte Konsistenz dementsprechend erreicht, wenn jede Interaktion vor ihrer Ausführung von jedem einzelnen Mitspieler bestätigt wird. Dieser Ansatz wird in der Echtzeitstrategie-Serie *Age of Empires* verfolgt und in Sektion 2.8.5 genauer erläutert.
Vollständige Synchronisation wird mit starken Einbußen auf Seiten der Interaktivität erkauft. Dementsprechend ist dieser Ansatz prinzipiell nur für Genres geeignet, die ohne das direkte Steuern von Spielfiguren auskommen. Diese stärkste Form der Konsistenz kommt in Teilbereichen aber auch in Actionspielen vor, wie beispielsweise beim Feuern von Projektilwaffen. Hierbei wird serverseitig nach Empfang des Client-Kommandos ein Projektil-Objekt erzeugt. Dementsprechend fliegt das Projektil erst los, nachdem clientseitig eine Antwort des Servers

empfangen wurde. Der Spieler muss diese Verzögerung beim Zielen mit einkalkulieren.

geeignet für:

- rundenbasierte Strategiespiele
- Echtzeit-Strategie

Local Lag

In Peer-to-Peer-Architekturen bietet sich mit *Local Lag* eine weitere Technik zur Erhaltung von Konsistenz an. Die grundlegende Idee dabei ist, die Ausführung aller lokalen Interaktionen um RTT/2 zu verzögern [Diot und Gautier (1999)]. Das Versenden der entsprechenden Nachricht an die anderen Teilnehmer erfolgt unverzüglich, und die Interaktion wird empfängerseitig nach Erhalt sofort ausgeführt. Im optimalen Fall ist die entsprechende Aktion somit nach RTT/2 (Peer-to-Peer-Architektur) bei allen Teilnehmern zeitgleich sichtbar. Weiterführende Variationen dieser Technik ergeben sich, wenn die Verzögerung nicht exakt auf RTT/2 gesetzt, sondern entsprechend den Spielanforderungen angepasst wird [M. Mauve und Effelsberg (2004)]. Der Entwickler berücksichtigt dabei die Folgen von Inkonsistenzen bezogen auf das konkrete Spiel, und wägt zwischen der Häufigkeit temporärer Inkonsistenzen und einer ausreichend schnellen Umsetzung der Interaktionen ab. Dazu wird zunächst ein minimaler Verzögerungswert definiert, der auf die höchste durchschnittliche Latenzzeit (RTT/2) zu den Mitspielern gesetzt wird. Dadurch wird erreicht, dass Inkonsistenzen nur noch bei Paketverlust oder starkem Jitter auftreten. Danach wird passend zum jeweiligen Spiel ein maximaler Verzögerungswert definiert, ab dem die Steuerung so träge wird, dass das Spielgefühl unzumutbar stark leidet. Arbeiten, die sich mit der Antwortzeit von Systemen befassen, geben einen maximalen Verzögerungswert von 80 bis 100 ms an, bevor eine Verzögerung vom Benutzer bemerkt wird [Shneiderman (1984)]. Dieser Wert kann je nach Spielgenre aber auch höher sein. Am Ende dieser Wertzuweisungen sind folgende Situationen möglich:

1. min. Verzögerungszeit $\leq$ max. Verzögerungszeit
   In diesem Fall kann die Verzögerungszeit auf einen Wert zwischen den beiden Extremwerten gesetzt werden

2. min. Verzögerungszeit > max. Verzögerungszeit
   Hier muss der Entwickler wohl oder übel seine Ansprüche in Sachen Konsistenz und/oder Interaktivität heruntersetzen und die Werte anpassen.

Local Lag kann das Auftreten von Inkonsistenzen nicht vollständig verhindern. Zum einen geht der Ansatz von symmetrischen Latenzen aus, die je nach Netzwerk nicht gegeben sind. Zum anderen können Inkonsistenzen durch Paketverlust und Jitter auftreten, falls zeitgleiche Aktionen der Mitspieler nicht innerhalb der Verzögerungszeit eintreffen. In diesem Fall

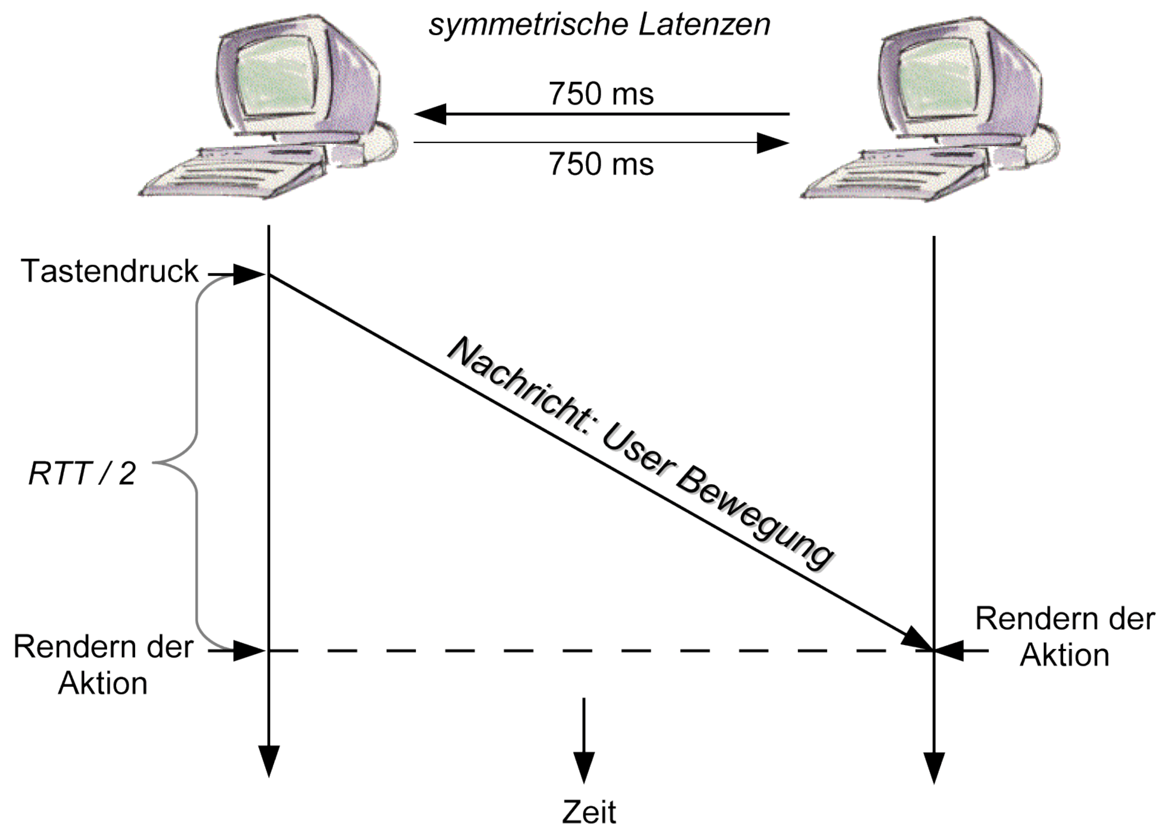


Abbildung 2.6: Local Lag

muss auf optimistische Synchronisationstechniken wie Time Warp (siehe 2.5.3) zurückgegriffen werden, um die Konsistenz wiederherzustellen. In Netzen mit konstanten Latenzen und geringem Paketverlust eignet sich Local Lag jedoch sehr gut für Spiele, die viel Wert auf Konsistenz legen, und trotzdem ein möglichst direktes Steuerungsverhalten bieten sollen. Hierbei sind an erster Stelle die Simulationen von Mannschaftssportarten zu nennen. Der große Vorteil von Local Lag ist dabei die übereinstimmende Position aller Spieler über alle Clients hinweg.
Für Client/Server-Architekturen relativiert sich der Vorteil von Local Lag, weil die Nachrichten aufgrund des doppelten Kommunikationsweges nicht um eine halbe, sondern um eine ganze RTT verzögert werden muss. Damit kann im Hinblick auf die zeitliche Verzögerung ebenso gut auf die Antwort des Servers gewartet werden, was dann wiederum der vollständigen Synchronisation (s. o.) entspräche.

Client Side Prediction

Als dritter und letzter Ansatz steht zur Wahrung der Konsistenz die *Client Side Prediction* zur Verfügung. Dieses Prinzip stellt eine maximale Interaktivität sicher, aber riskiert gleichzeitig die größten Abweichungen unter den einzelnen Spielersichten. Der Begriff *Client Side Prediction* kam erstmals im Jahr 1996 im Zusammenhang mit dem 3D-Shooter *Quake* auf, das in Sektion 2.8.1 besprochen wird. Der Ansatz entspricht dem im militärischen Bereich verfolgten Prinzip, dass sich die Vernetzung mehrerer Simulatoren nicht auf das Steuerungsverhalten der Entitäten auswirken darf. Stattdessen versuchen die Teilnehmer, sich einem gemeinsamen Zustand der virtuellen Welt möglichst gut anzunähern. Dieser Ansatz nimmt Inkonsistenzen in Kauf, und versucht lediglich, diese Abweichungen möglichst gering zu halten. Welche Probleme durch Inkonsistenzen entstehen können, betrachten wir im folgenden Unterkapitel.

2.5.2 Probleme durch Inkonsistenzen

Die Nutzung von Local Lag und Client Side Prediction erschwert die Interaktion zwischen den Teilnehmern, da eine inkonsistente Sicht der Entitätspositionen unter den Spielern möglich oder sogar wahrscheinlich ist. Im Folgenden werden diesbezüglich typische Probleme betrachtet, die in Multiplayerspielen regelmäßig auftauchen.

Zielen

Abweichungen in der dargestellten Position erschweren in actionorientierten Spielen das Zielen auf verteilte Spielobjekte. Feuert ein Teilnehmer seine Waffe auf eine entfernte Entität, verfehlt er möglicherweise aus Sicht eines anderen Teilnehmers, obwohl die Entität in seiner Darstellung exakt im Zielkreuz war. Es stellt sich also die Frage, wessen Sicht als Referenz für die Trefferauswertung herangezogen werden soll. In der im kommerziellen Bereich vorherrschenden Client/Server-Architektur wurde anfangs stets der Einfachheit halber die serverseitige Sicht gewertet. Im Falle von aktuellen 3D-Shootern der Serien Counter-Strike und Unreal Tournament entscheidet bei Instant-Hit-Waffen mittlerweile die clientseitige Sicht im Moment des Feuerns über die genaue Trefferzone. Aufgrund der Gefahr des Cheatings ist dazu eine relativ aufwendige Technik namens *Lag Compensation* notwendig. Das Implementieren dieser Technik wird in Kapitel 2.8.2 behandelt.

Kollisionen

Inkonsistente Sichten auf die Spielwelt machen es schwierig, sich auf Kollisionen zu einigen. Das Problem ist mit dem der Trefferauswertung vergleichbar, wirkt sich aber stärker auf den

Spielfluss aus, da die Objekte nach einer Kollision auf eine bestimmte Position zurückgesetzt werden müssen. Ein Lösungsansatz ist, aus den verschiedenen Sichten der Teilnehmer einen Kompromiss zu errechnen, der dann als Referenz gilt. Diese Technik namens *Rendezvous-Algorithmus* wird in Kapitel 2.5.3 vorgestellt. Sie spielt aufgrund ihrer Komplexität und der wenig nachvollziehbaren Ergebnisse in der Praxis jedoch zur Zeit keine Rolle. Der praktikablere Ansatz ist es, einen der Teilnehmer als autoritative Instanz zu definieren, dessen Sicht auf die Spielwelt stets als Referenz dient. Da in aktuellen kommerziellen Echtzeitspielen die Client/Server-Architektur vorherrscht, übernimmt diese Rolle der Spieleserver, zumal dieser normalerweise auch für die Berechnung der physikalischen Auswirkungen zuständig ist, die durch die Kollision entstanden sind. Da der Server den Mittelpunkt der Kommunikation darstellt, bietet er bei in etwa gleich großen Latenzen der Teilnehmer automatisch den besten Mittelwert zwischen allen Spielersichten. Stellt der Server eine Kollision fest, berechnet er neue Positionen der kollidierten Spielobjekte und sendet entsprechende Update-Nachrichten an die Clients. Weicht die Sicht des Clients von der des Servers stark ab, führt diese Korrektur in der Client-Darstellung zu störenden Sprüngen der Spielobjekte. Je höher die Latenz ist, desto stärker und zahlreicher treten diese Sprünge auf, und desto mehr leidet der Spielspaß.

Manipulation verteilter Objekte

Die Latenz führt in Netzwerkspielen auch zu klassischen Problemen aus dem Bereich der verteilten Systeme. Man stelle sich beispielsweise eine Spielsituation vor, in der zwei Spielfiguren in einer Rollenspielwelt beim gleichen Händler einkaufen, und beide in etwa zur gleichen Zeit den gleichen Gegenstand erwerben wollen, der aber nur als Einzelstück beim Händler verfügbar ist. Als Lösung hierfür bieten sich optimistische Synchronisationsalgorithmen an, die in 2.5.3 erläutert werden.

Wer-erschießt-wen-Problem

In ereignisorientierten verteilten Simulationen, zu denen auch die Echtzeitspiele gehören, spielt die Abarbeitungsreihenfolge der Ereignisse eine erhebliche Rolle. Im Bereich der Multiplayerspiele ist die richtige Reihenfolge der Events vor allem in Situationen wichtig, in denen die Reaktionszeit des Spielers über Sieg oder Niederlage entscheidet. Ein zentraler Problemfall ist die folgende Situation, die in 3D-Shootern häufig auftritt. Wir gehen dabei von einer Situation aus, in der drei Spieler aufeinander treffen, die mit Instant-Hit-Waffen ausgerüstet sind, deren Treffer sofort tödlich ist.

- Spieler A feuert auf Spieler B
- B feuert fast gleichzeitig auf C, war aber etwas langsamer als A

- B hat eine geringere Latenz als A, sodass sein Feuer-Kommando zuerst beim Server eintrifft
- $\Longrightarrow$ Der Tod von C ist unzulässig, weil A nicht mehr hätte feuern dürfen.

Aufgrund der Netzwerklatenzen ergibt sich also bei der sofortigen Ausführung des nächsten eingegangenen Events das Problem, dass die Simulationslogik nicht sicher sein kann, ob das Ausführen dieses Events zulässig ist. Es muss also eine Möglichkeit geschaffen werden, die zeitliche bzw. logische Reihenfolge von Events wiederherzustellen. Dies ist ein zentrales Problem aus dem Bereich der verteilten Systeme, für das mehrere Lösungsansätze existieren, die sich grob in *konservative* und *optimistische* Algorithmen unterteilen. Im Folgenden prüfen wir die Tauglichkeit beider Ansätze für den Einsatz in verteilten Echtzeitspielen.

2.5.3 Synchronisationsalgorithmen

Konservative Synchronisationsalgorithmen

Die konservativen Algorithmen basieren auf der Herstellung einer kausalen Ordnung der Events mit Hilfe des von Lamport erdachten Prinzips der logischen Zeitstempel [Lamport (1978)]. Diese Ordnung stellt sicher, dass die kausale Konsistenz im Sinne einer seriell-äquivalenten Abarbeitung gewahrt wird. In unserem Kontext kommt jedoch der Echtzeit-Aspekt als zusätzlicher Faktor hinzu. Ereignisse sind nicht nur an ihren kausalen Zusammenhang gebunden, sondern auch an einen bestimmten Zeitpunkt. Dieser zeitliche Zusammenhang wird von konservativen Algorithmen nicht berücksichtigt, da das Fortschreiten der Simulationszeit nicht an das Fortschreiten der reellen Uhrzeit gekoppelt ist. Dies führt in Echtzeitspielen zu erheblichen Einschränkungen der Interaktivität, da Prozesse aufeinander warten und blockieren können. Alle konservativen Algorithmen benötigen außerdem für eine gute Performance einen ausreichend hohen *Lookahead*, d.h. sie müssen anhand von kontextspezifischen Fakten sozusagen ein Stück in die Zukunft schauen können, um für ein bestimmtes Ereignis einen Rollback ausschließen zu können [Fujimoto (2000a)]. Beispielsweise kann ein Fluglotsen-Simulator für jede Flugstrecke eine bestimmte Mindestflugdauer voraussetzen, und so eine Landung innerhalb einer bestimmten Zeitspanne ausschließen. Konservative Algorithmen werden im Bereich der verteilten analytischen Simulationen eingesetzt, aber eignen sich aufgrund der Unvorhersagbarkeit der Spieleraktionen und des Echtzeitkontextes nicht für die Implementierung in DVEs.

Optimistische Synchronisationalgorithmen

Einen alternativen Ansatz zur Wahrung der kausalen Ordnung bieten die optimistischen Synchronisationsalgorithmen. Hier wird zunächst blind davon ausgegangen, dass die Aus-

führung des nächsten Ereignisses zulässig ist. Im Falle von Konflikten wird die Ausführung unzulässiger Events dann im Nachhinein rückgängig gemacht.

Time Warp

Jefferson's Time Warp ist der erste und bis heute bekannteste optimistische Synchronisations-Algorithmus in ereignisorientierten Simulationen. Bei Anwendung des Time Warp Algorithmus' werden die abgearbeiteten Events und ursprünglichen Zustände in einer Queue abgelegt, damit im Falle eines Konflikts später ein Rollback möglich ist. Es ist im optimistischen Ansatz also jederzeit möglich, dass eine Nachricht mit einem Event eintrifft, der einen niedrigeren Zeitstempel hat als der soeben ausgeführte. Solche verspäteten Events werden in Time Warp als *Straggler Messages* bezeichnet (siehe Abb. 2.7). Events mit höherem Zeitstempel, die vor einer Straggler Message ausgeführt wurden, wurden somit unrechtmäßig ausgeführt, weil möglicherweise das Kausalitätsprinzip verletzt wurde. Der Time Warp Algorithmus löst dieses Problem, indem er die unzulässige Abarbeitung der verfrüht ausgeführten Ereignisse rückgängig macht, und danach alle betroffenen Ereignisse nochmals vollständig in der Reihenfolge ihrer Zeitstempel ausführt. Es ist also ein Mechanismus nötig, der die beiden folgenden möglichen Auswirkungen von Ereignisverarbeitungen rückgängig macht:

1. Veränderung von Zustandsvariablen des Systems
2. Erzeugen von neuen Ereignissen und das Versenden von neuen Nachrichten

<u>Rollback von Zustandsvariablen:</u>
Der Rollback von Zustandsvariablen kann auf zwei Arten implementiert werden:

1. copy state saving
   Der lokale Time Warp Prozess macht eine lokale Kopie aller Zustandsvariablen, die durch Ereignisse verändert werden können. Zu jedem Event existiert ein kompletter Snapshot aller Zustandsvariablen, die durch Events verändert werden können. Bei Eintreffen einer Straggler Message wird der gesamte Block zurückkopiert.
2. incremental state saving
   Statt alle Variablen komplett zu sichern, wird pro Ereignis immer nur je ein Zeiger auf jede veränderte Variable gesichert, sowie ihr Wert vor Ausführung des Ereignisses gesichert. Trifft eine Straggler Message ein, wird jedes unzulässig durchgeführte Ereignis nun einzeln rückgängig gemacht, und der jeweils vorherige Wert wiederhergestellt.

Welches der beiden Verfahren effizienter ist, hängt davon ab, wie viele Werte im Durchschnitt pro Ereignis geändert werden. Viele Time Warp Implementierungen benutzen auch eine Kombination aus beiden Techniken. In jedem Fall muss zu jedem Ereignis eine bestimmte

Datenmenge an Zustandsinformationen verwaltet werden. Die entsprechende Datenstruktur innerhalb des lokalen Prozesses wird bei Time Warp als *state queue* bezeichnet. Die eigentlichen Events, egal ob sie schon abgearbeitet sind oder nicht, werden in einer Datenstruktur namens *input queue* verwaltet.

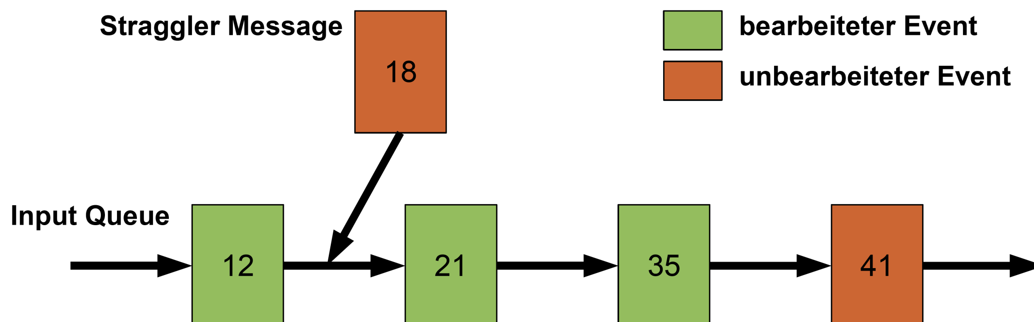


Abbildung 2.7: Input-Queue in Time Warp bei Ankunft einer Straggler Message

Rollback von Nachrichten:
Wie kann man das Senden von Nachrichten rückgängig machen? Time Warp beinhaltet dazu das Konzept der *Anti-Messages*. Der Name ist an den Begriff der Antimaterie aus dem Bereich der Partikelphysik angelehnt. Kommt ein Materie-Atom mit einem Antimaterie-Atom in Berührung, neutralisieren sich die beiden gegenseitig und verschwinden. Analog dazu stellt eine Anti-Message ein negatives Pendant zu einer konkreten Message dar. Eine Anti-Message ist eine identische Kopie einer Message, bei der aber ein zusätzliches Flag gesetzt ist, das sie als Anti-Message kennzeichnet. Wird eine Message zusammen mit ihrer Anti-Message in einer Queue abgelegt, heben sich die beiden auf und ihr Speicher wird freigegeben. Um das Versenden einer Nachricht rückgängig zu machen, muss ein lokaler Prozess also nur die dazugehörige Anti-Message versenden. Das impliziert, dass der Prozess eine Liste mit allen Events führen muss, die er verschickt hat, um später bei Bedarf daraus entsprechende Anti-Messages generieren zu können. Dazu verwaltet jeder Prozess intern eine Datenstrukur namens *output queue*. Wird eine Message versendet, legt der Prozess die dazugehörige Anti-Message in seiner output queue ab. Wird für ein Ereignis ein Rollback ausgelöst, werden seine zugehörigen Anti-Messages versendet. Empfängt ein Prozess eine Anti-Message, löst er einen Rollback bis zu dem Zeitpunkt vor Abarbeiten des entsprechenden Events aus, wobei er seinerseits Anti-Messages verschicken kann. Durch den rekursiven Prozess der Rollbacks kombiniert mit dem Versenden von Anti-Messages, werden sämtliche unzulässig ausgeführten Berechnungen aller beteiligten Prozesse rückgängig gemacht.

Local Rollback

Das Konzept der Anti-Messages bringt verschiedene Probleme wie beispielsweise die Gefahr von endlosen Rollback-Zyklen mit sich, die nur durch teils aufwändige Mechanismen verhindert werden können, und mit Performance-Einbußen einhergehen. Mit *Local Rollback* existiert eine Time Warp Variante, die auf Anti-Messages verzichtet. Stattdessen wird der Versand von Messages zurückgehalten, bis ein Rollback des entsprechenden Ereignisses ausgeschlossen werden kann.
Es ergibt sich also das Problem zu entscheiden, ab welchem Zeitpunkt ein Rollback eines Ereignisses definiv ausgeschlossen werden kann. Diese zeitliche Untergrenze wird in Time Warp als *Global Virtual Time* (GVT) bezeichnet. Da Rollbacks nur durch Straggler Messages ausgelöst werden können, die wiederum durch das Abarbeiten von Events erzeugt werden, ergibt sich die GVT durch den niedrigsten Zeitstempel aller unbearbeiteten und teilweise bearbeiteten Messages in einer globalen Momentaufnahme des Time Warp Systems. Das Erstellen einer solchen globalen Momentaufnahme eines verteilten Systems stellt aufgrund der Netzwerklatenzen ein nicht-triviales Problem dar.
Im Bereich der DVEs können wir auf eine exakte Ermittlung der GVT verzichten, da zum einen ein Rollback generell nur innerhalb einer bestimmten Zeitspanne tolerierbar ist, ohne eine Irritation oder Frustration der Spieler zu riskieren. Zum anderen können wir uns die hohe Interaktionsrate zu Nutze machen, die in Echtzeitspielen für eine regelmäßige und relativ hochfrequente Erzeugung von Ereignissen und somit von Datenpaketen sorgt. Ein Rollback des Ereignisses E_x zum Zeitpunkt t kann ausgeschlossen werden, sobald von allen Teilnehmern ein Datenpaket mit einem höheren Zeitstempel als t eingegangen ist, und dabei kein Ereignis übermittelt wurde, dass E_x verhindert hat. Es bietet sich deshalb an, die Client-to-Server Datenpakete in einer konstanten und hohen Frequenz zu versenden, um diese Zeitspanne möglichst niedrig und konstant zu halten.

Bewertung:
Bezogen auf unser „wer-erschießt-wen“-Beispiel ergeben sich nun zwei Möglichkeiten:

1. Möglichkeit: Anti-Messages
   Der Server propagiert die B-erschießt-A sofort nach Erhalt an die Clients weiter. Empfängt er kurz danach eine Straggler Message, verschickt der Server die Anti-Message zu *B-erschießt-A*. Dieser Ansatz führt dazu, dass Spieler A den Tod seiner Spielfigur auf dem Bildschirm sieht, die dann kurz darauf plötzlich wiederbelebt wird, nachdem der Rollback durch die Anti-Message ausgelöst wird.

2. Möglichkeit: Local Rollback
   Der Server propagiert das Ereignis *B-erschießt-A* solange nicht an die Clients, bis dass er von jedem Spieler eine Nachricht mit einem höheren Zeitstempel als *B-erschießt-A*

> erhalten hat. Sobald dies der Fall ist, kann der Server einen Rollback ausschließen und A verbindlich für tot erklären. Erhält der Server stattdessen eine Straggler Message, so genügt es, das Ereignis durch einen serverinternen Rollback rückgängig zu machen. In jedem Fall wird das Ereignis erst dann propagiert, sobald von jedem Spielen Nachrichten mit ausreichend hohem Zeitstempel eingegangen sind.

In aktuellen kommerziellen Multiplayerspielen wird aus mehreren Gründen der Local-Rollback-Ansatz bevorzugt. Der Hauptgrund ist sicherlich die Irritation des Spielers, die durch einen Rollback entsteht, zumal Inkonsistenzen nach Art des Wer-erschießt-wen-Problems in 3D-Shootern relativ häufig auftreten. Im Extremfall geht der Spieler zu Boden, und wird dann im Falle eines Rollbacks kurz danach auf scheinbar wundersame Weise wiederbelebt.
Der Nachteil von Local Rollback ist, dass durch das Zurückhalten der Nachrichten eine zusätzliche Verzögerung entsteht, die im Falle von wiederholtem Paketverlust theoretisch beliebig hoch sein kann. In der Praxis bietet sich jedoch die Implementierung einer zeitlichen 'Schmerzgrenze' an, ab der ein Rollback nicht mehr tolerierbar ist, und die daraus möglicherweise entstehenden Ungerechtigkeiten zu Gunsten eines flüssigeren Spielerlebnisses als „höhere Gewalt“ hinzunehmen.

Bucket Synchronisation Mechanism

Bucket-Synchronisation ist eine Technik, die mit dem 1997 erschienenen Freeware-Spiel *MiMaze* [Gautier und Diot (1998)] vorgestellt wurde. MiMaze ist ein Echtzeit-Multiplayerspiel, mit dem die Macher beweisen wollen, dass eine dezentrale Kommunikation für Online Spiele besser geeignet ist, als die verbreitete Client/Server-Architektur (siehe 2.6.2.1). Die beiden Hauptelemente des MiMaze-Netzwerkcodes sind ein Multicast-Kommunikations-System basierend auf RTP/UDP/IP, und ein verteilter Synchronisationsalgorithmus. Der Algorithmus ermöglicht, dass alle Spieler trotz unterschiedlicher Roundtrip-Zeit zu jedem Zeitpunkt den annähernd gleichen Zustand der Welt sehen. Da kein Server als Zeitgeber dient, ist eine absolute Uhrensynchronisation zu einer universellen Uhrzeit unter allen Teilnehmern notwendig.

<u>Funktionsweise:</u>
Die Spielzeit wird in Perioden fester Länge eingeteilt, bei MiMaze in 25 pro Sekunde. Zu jedem Zeitabschnitt wird eine Speichereinheit („Bucket“) eingerichtet. Die Buckets werden nach einer festen Zeitspanne (*allgemeine Verzögerung*) von beispielsweise 100ms abgearbeitet, um die Spielwelt darzustellen. Jeder Spieler sammelt Datenpakete („ADUs“) von anderen Spielern, sodass optimalerweise in jedem Bucket zum Zeitpunkt der Darstellung je eine ADU von jedem Spieler existiert. Kommt eine ADU nicht mehr rechtzeitig an, wird sie verworfen. Zur Darstellung des entsprechenden Spielers wird dann stattdessen die letzte ADU benutzt, die von dem Spieler erhalten wurde. Die Bewegung des lokalen Spielers wird

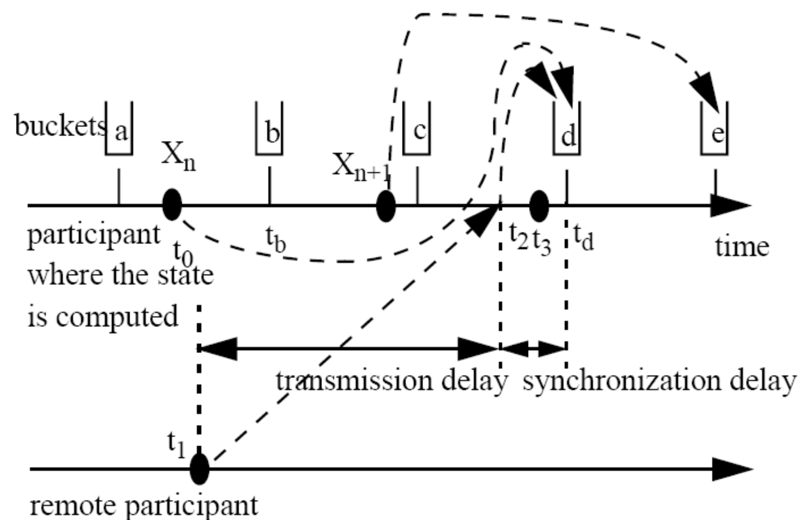


Abbildung 2.8: Bucket-Synchronisation [Gautier und Diot (1998)]

ohne Verzögerung angezeigt.

Vorteile:

- Minimierung der Latenz durch direkte Kommunikation der Teilnehmer („Peer-to-Peer“)
- kein Single-Point-Of-Failure
- bessere Skalierbarkeit, da sich die Netzlast auf alle Teilnehmer verteilt

Nachteile:

Die aktuelle Version von MiMaze bietet zwar mittlerweile *Dead Reckoning* (siehe 2.7.1) zur Verringerung des Darstellungsfehlers bei verspäteten ADUs, jedoch noch keine Fehlerkorrektur bei kritischen Events. So bleibt MiMaze die Antwort auf die Frage schuldig, wie mit Paketverlust bei kritischen Events (wie z.B. Drücken der Feuertaste) umgegangen wird. Was passiert im unter 2.5.2 beschriebenen Szenario, falls das Opfer den Event mit dem eigenen Todesschuss nicht rechtzeitig erhalten hat? Inkonsistenzen dieser Art sind mangels zentraler Autorität nur sehr schwer aufzulösen. Weitere Probleme ergeben sich bei der Einstellung der allgemeinen Verzögerung, die sich immer nach dem schwächsten Glied in der Kette (sprich: der Verbindung mit der höchsten Latenz) richtet, und ggf. angepasst werden muss.

Rendezvous Mechanismus

Rendezvous ist ein eher experimenteller Konsistenz-Algorithmus für Peer-to-Peer-Architekturen (siehe Abschnitt 2.6.2.2), der 2004 von zwei Studenten des *Computing Department at Lancaster*[3] vorgestellt und in einem verteilten Fußballspiel namens „Knockabout“ implementiert wurde [Chandler und Finney (2005)]. Rendezvous wurde speziell für die unzuverlässigen Bedingungen in mobilen Netzen konzipiert, und erlaubt einen gewissen Grad an Inkonsistenz zwischen den Spielzuständen aller Peers. Die grundlegende Idee dabei ist, dass sich der lokale Spielzustand eines Peers den Spielzuständen der anderen Peers annähert. Die einzelnen Spieler tauschen ihre lokalen Sichten aus, und einigen sich auf einen gemeinsamen Spielzustand, der den besten Kompromiss aus den Sichten aller Spieler darstellt. Dazu verschickt jeder Peer in Intervallen eine serialisierte Version seines gewünschten Zielzustandes („Target State“) an die anderen. Anhand von festen Regeln („Adaptation Rules“), die Nichtspielerobjekte beeinflussen und sogar die Spielregeln geringfügig verändern können, versucht jeder Peer, den eigenen Target State auf den der anderen anzugleichen. Da in der Zwischenzeit ständig neue Inkonsistenzen auftreten, ändert sich dieser Target State ständig, aber wird praktisch nie erreicht.

Bewertung:
Es war uns nicht möglich, zu Evaluationszwecken im Internet eine lauffähige Version des „Knockabout“-Spiels zu finden, geschweige denn Auszüge des Sourcecodes. Die Beschreibung des Algorithmus lässt viele Implementierungsdetails offen, und der Webauftritt einer geplanten Rendezvous-API wurde seit 2005 nicht mehr aktualisiert. Auch wenn die Ausführungen der Entwickler recht optimistisch klingen, sind wir eher skeptisch, was die Wahrung der Fairness und der Spielbarkeit betrifft. Außerdem scheint fraglich, ob derart komplexe Berechnungen auf mobilen Endgeräten noch praktikabel sind.

2.5.4 Zeitsynchronisation

Durch die Verwendung von Zeitstempeln ergibt sich die Notwendigkeit einer relativen Uhrensynchronisation zwischen allen Teilnehmern. Alle Ereignisse müssen jeweils einem festen Zeitpunkt auf einer gemeinsamen Zeitachse zugeordnet werden, sodass eine gemeinsame totale Ordnung der Ereignisse über alle Spieler hinweg erreicht werden kann. Wir unterschieden im Bereich der verteilten Systeme verschiedene Uhrzeiten:

Systemzeit: Jedes aktuelle Endgerät verfügt über einen internen Zeitgeber. Dieser zeigt bei korrekter Justierung die Anzahl der *Ticks* an, die seit dem willkürlich gewählten Zeitpunkt 1970-01-01 00:00:00 vergangen sind. Die Auflösung der Ticks ist von System zu

[3] http://www.comp.lancs.ac.uk/ angie/rendezvous/main.html

System unterschiedlich. Für mobile Endgeräte schreibt der aktuelle Standard MIDP2.0 eine Genauigkeit von einer Millisekunde vor.

Serverzeit: die Systemzeit des Servers

Weltzeit: Eine koordinierte Uhrzeit, die im Gegensatz zur Lokalzeit die Zeitverschiebung berücksichtigt, und die somit für den gesamten Globus eindeutig ist. Der heutige Standard ist *Universal Time Coordinated* (UTC). Um sich auf UTC zu synchronisieren, wird das *Network Time Protocol* (NTP) benutzt, womit eine Genauigkeit von ca. 1 - 50 ms erreicht wird [Mills (1992)].

Eine Synchronisation auf die Weltzeit ist in Multiplayerspielen bei einer Client/Server Architektur nicht nötig, da keines der zu erwartenden Ereignisse einen Bezug zur aktuellen Weltzeit haben wird. Das ist in militärischen Simulationen anders, da dort nicht nur Menschen, sondern auch militärische Geräte agieren. In unserem Anwendungsgebiet reicht als gemeinsame Referenz die Systemzeit des Servers aus. Es ist außerdem nicht nötig, die Systemuhr tatsächlich der des Servers anzupassen, da zum zeitlichen Einordnen der Ereignisse nur die Differenz bekannt sein muss. Zur zentralisierten Uhrensynchronisation existieren zwei verschiedene Ansätze:

1. zentralisierter Push Algorithmus

2. zentralisierter Pull Algorithmus

zentralisierter Pull Algorithmus

Der zentralisierte Pull-Algorithmus entspricht der folgenden Vorgehensweise: Ein Client schickt eine Zeitwert-Anfrage zum Server, und wartet auf eine Antwort. Wir bezeichnen den Zeitpunkt auf der Systemuhr des Clients, zu dem er die Anfrage abschickt, als T_S. Die Antwort trifft zum Zeitpunkt T_R ein. Der vom Server gesendete Zeitwert beträgt T, und die Latenz, um diese Information vom Server zum Client zu senden, nennen wir L. Demzufolge beträgt der Zeitpunkt, zu dem die Antwort empfangen wird, $T+L$, aber L ist unbekannt. Unter der Annahme, dass die Dauer des Netzwerktransfers für Anfrage und Antwort ungefähr gleich ist, kann der Client den Zeitpunkt des Eintreffens der Antwort in Serverzeit TS_R mit folgender Formel errechnen:

$$TS_R = T + (T_R - T_S - I)/2$$

Hierbei bezeichnet I die Zeit, die der Server zum Verarbeiten der Anfrage braucht. Die Differenz D zwischen T_R und TS_R stellt die gesuchte Abweichung der clientseitigen Systemzeit zur Serverzeit dar (vgl. Abb. 2.9).
In der Praxis ist L aufgrund von schwankender Prozessor- und Netzwerklast nicht symmetrisch, daher ist es ratsam, mehrere Anfragen zu senden, dabei Extremwerte zu ignorieren,

und aus den errechneten Differenzen einen Mittelwert zu bilden („Algorithmus von Christian“, siehe [Cristian (1989)]).

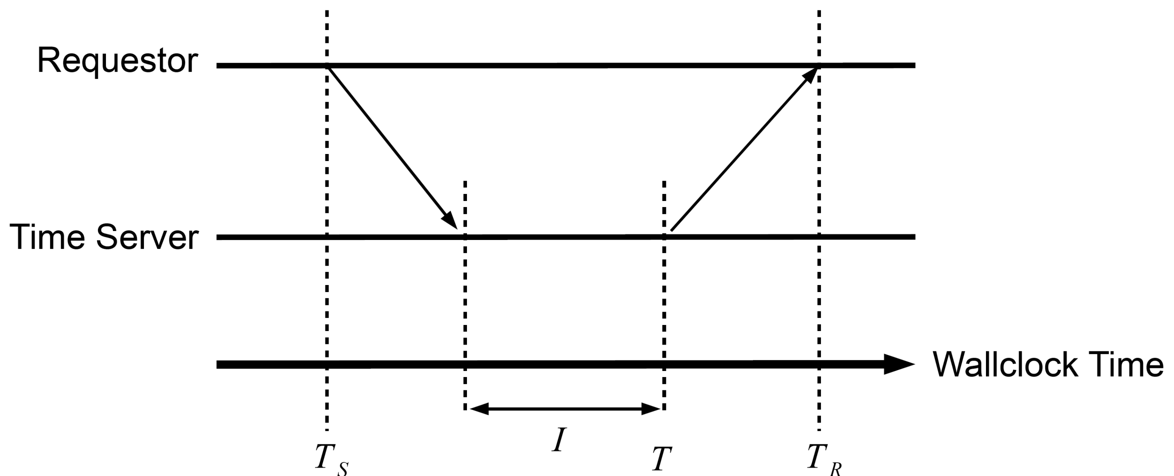


Abbildung 2.9: zentralisierter Pull Algorithmus

zentralisierter Push Algorithmus

In diesem Ansatz schickt der Server ohne explizite Anfrage der Clients in regelmäßigen Abständen seine aktuelle Uhrzeit. Die Clients müssen L aufgrund von vorherigen Messungen schätzen, da L nicht ohne das Generieren von zusätzlichen Nachrichten angenähert werden kann.

Synchronisationsfehler

In der Praxis werden die Zeitgeber zweier Systemuhren nie vollständig synchron laufen, sondern immer weiter auseinander driften. Läuft die Uhr eines Clients beispielsweise schneller als die des Servers, und legt man trotzdem einen konstanten Wert für D zugrunde, so werden die vom Server übermittelten Ereignisse nach und nach zeitlich immer früher eingeordnet, als sie tatsächlich passiert sind. Es ist deshalb notwendig, D regelmäßig zu aktualisieren. Wie kurz diese Intervalle sein müssen, ist aufgrund der unüberschaubaren Anzahl der mobilen Endgeräte schwer vorherzusagen. Gehen wir als stärkstem zu erwartenden Synchronisationsfehler von einer Minute Abweichung pro Tag aus, ergibt sich eine Abweichung von 2,5 Sekunden pro Stunde, bzw. 41.66 ms pro Minute. Basierend auf diesen Daten sollte also eine Resynchronisation alle 10 Sekunden ausreichend sein, um eine Bevor- oder Benachteiligung einzelner Spieler auf lange Sicht auszuschließen.

Asymmetrische Latenzen

In den beschriebenen Algorithmen kann nur die Roundtrip-Time (RTT) gemessen werden, womit die vollständige Zeit vom Absenden der Anfrage bis zum Erhalt der Antwort bezeichnet wird. Die Genauigkeit der Synchronisation hängt also stark von der Symmetrie der Netzwerklatenzen ab, die in Wide Area Netzwerken oft nicht gegeben ist [Fujimoto (2000b)]. In wieweit in mobilen Netzen die Nachrichten vom Server zum Client schneller oder langsamer übermittelt werden als die Nachrichten vom Client vom Server, übersteigt den Rahmen dieser Arbeit. Dazu müsste ein Testsystem eingerichtet werden, das mit Hilfe einer zusätzlichen direkten Verbindung zwischen mobilem Client und Server exakt den Zeitpunkt zwischen Absenden und Empfang einer einzelnen Nachricht messen kann. Folgende Ergebnisse wären denkbar:

1. schwankende Asymmetrie innerhalb einer Verbindung
   In diesem Fall wird ein Spieler langfristig bevor- bzw. benachteiligt, falls die relative Differenz der Systemzeiten nur anhand einer einzelnen Messung erfolgt, und diese Messung stark vom Durchschnittswert abweicht. Die Differenz sollte deshalb über mehrere Messungen gemittelt, und Extremwerte dabei nicht berücksichtigt werden.

2. konstante Asymmetrie innerhalb einer Verbindung
   Dieser Fall ist gegeben, falls bei einem Spieler Client-to-Server jedes Mal schneller bzw. langsamer unterwegs sind, als Server-to-Client-Pakete. In diesem Fall wird ein Spieler langfristig bevor- bzw. benachteiligt, was auch durch Mitteln der Differenz nicht verhindert werden kann. Die einzig denkbare Möglichkeit in diesem Fall ist ein Kompensieren der Asymmetrie durch einen konstanten Faktor, der auf Heuristiken basiert.

3. konstante Asymmetrie über alle Verbindungen hinweg
   In diesem Fall kommen Client-to-Server Pakete bei allen Spielern schneller bzw. langsamer an, als Server-to-Client-Pakete. Somit würden alle Spieler den gleichen Vor- bzw. Nachteil erhalten, wodurch ein gerechter Spielablauf gewährleistet ist.

4. schwankende Asymmetrie bei allen Verbindungen
   Ist die bei allen Teilnehmern schwankend und nicht vorhersagbar, empfiehlt sich ebenfalls ein Mitteln der relativen Differenz über mehrere Messungen hinweg. So wird erreicht, dass ein Spieler einen Vor- bzw. Nachteil nur jeweils von Ereignis zu Ereignis erhält, und keinen konstanten Vor-/Nachteil über das gesamte Spiel hinweg. So kann der Faktor Zufall zwar nicht eliminiert werden, gleicht sich aber im Laufe eines Spiels immer mehr aus.

2.6 Minimierung der Latenz

Die Netzwerklatenz ergibt sich in erster Linie aus den Zeiten, die zum Weiterleiten der Datenpakete durch das zugrundeliegende Netzwerk benötigt werden. Für den Anwendungsentwickler bieten sich deshalb relativ wenige Möglichkeiten der Optimierung. Eine davon ist die Wahl des zur Datenübertragung genutzten Transportprotokolls. Seit MIDP2.0 sind Datagramm- und Serversockets Teil des Java-Standards, was letztlich auf die Transportprotokolle TCP und UDP des TCP/IP-Protokollstacks abgebildet wird. Eine andere Optimierungsmöglichkeit zur Minimierung der Netzwerklatenz bietet sich in der Wahl der Netzwerktopologie. Hier stehen die beiden grundsätzlichen Ansätze der Client/Server-Architektur und der Peer-to-Peer-Architektur zur Auswahl.

2.6.1 Wahl des Transportprotokolls

Reliable UDP

UDP hat sich als das Protokoll der Wahl für den Echtzeitdatenaustausch bei Actionspielen etabliert. Sogenannte „Twitchgames“, Spiele bei denen die Tastatureingabe eine sofortige drastische Veränderung der Spielfigur bewirken und eine extrem schnelle Interaktion zwischen den Spielern besteht, wie z.B. First Person Shooter, sind darauf angewiesen, dass die Nachrichten auf schnellste Weise übertragen werden. TCP ist aufgrund seiner umfangreichen Korrekturmaßnahmen ein eher ungeeignetes Protokoll für Echtzeit-Actionspiele. Der für die garantierte Zustellung und Reihenfolgeerhaltung der Pakete getätigte Verwaltungsaufwand macht TCP wesentlich träger und unflexibler als UDP. Wenn z. B. ein Paket in falscher Reihenfolge empfangen wurde, hält TCP alle weiteren empfangenen Pakete zurück, bis das Paket mit der erwarteten Sequenznummer erhalten wurde. Dies ist in Echtzeitspielen kontraproduktiv, da der überwiegende Teil der Daten periodische Zustandsaktualisierungen sind, die möglichst schnell verarbeitet werden müssen. Des weiteren führt TCP eine Überlaststeuerung durch: Falls die Leitung überlastet ist und ein Paket nicht zugestellt werden kann, hört TCP irgendwann auf, das Paket zu versenden. Aufgrund von periodisch versendeten Testpaketen wird festgestellt, ob die Leitung wieder frei ist. Ist dies der Fall, fängt TCP schrittweise wieder an, die eigentlichen Nutzdaten zu versenden. Dieser „slow start“ Algorithmus sorgt dafür, dass extrem hohe Latenzen entstehen können. UDP gibt im Gegensatz keine Auskunft darüber, ob ein Paket den Empfänger erreicht hat, und es garantiert auch nicht, dass Pakete in der Reihenfolge, in der sie versendet wurden, ankommen Lincroft (1999). Da jedoch bei „Twitchgames“ ständig neue Daten in sehr kurzen Abständen anfallen, ist in den meisten Fällen ein möglicher Paketverlust irrelevant. Auch gelegentliche Reihenfolgefehler sind tolerierbar, da Zustandsupdates normalerweise nur wenige Bytes umfassen und nicht in mehrere Datenpakete aufgeteilt werden müssen. UDP hat weiterhin den Vorteil, dass es im Gegensatz zu TCP ein verbindungsloses Protokoll ist, wodurch kein zusätzlicher Aufwand

für den Verbindungsauf- und abbau entsteht. Das Standard UDP wird also für Spielnachrichten wie Bewegungsdaten, die permanent aktualisiert werden, und die auch ab und an verloren gehen oder vertauscht werden können, vollkommen ausreichend sein. In Echtzeit-Actionspielen fallen jedoch auch Nachrichten an, deren Zustellung garantiert werden muss, wie z.B. Schüsse und daraus resultierende Treffer. Daher ist es nötig, UDP um einige sinnvolle Funktionalitäten zu erweitern, die TCP bereits beinhaltet. Diese müssen jedoch im Hinblick auf die Echtzeitanforderungen optimiert werden.

ENet

ENet ist eine Netzwerkschicht, die auf UDP aufsetzt und Eigenschaften von TCP anbietet, die im Hinblick auf die Erfordernisse von Echtzeitspielen optimiert sind. Ursprünglich wurde ENet entworfen, um die Echzeitdatenübertragung im 3D-Shooter *Cube*[4] zu ermöglichen. ENet ist frei verfügbar [Salzman (2002)] und wurde von Exit Games für die Reliable-UDP-Schicht *NNet* der Neutron-Echtzeit-Plattform angepasst (siehe Abschnitt 3.1). ENet bietet folgende Funktionen:

- Verbindungsverwaltung: Es kann eine logische Verbindung zu einem Endpunkt aufgebaut und verwaltet werden. Bestehende Verbindungen werden mittels sporadisch versendeter Pings überprüft und aufrechterhalten.

- Zuverlässigkeit: Auf Wunsch können die UDP Pakete zuverlässig versendet werden, d. h. ihr Erhalt muss mit einem ACK bestätigt werden. Sollte für ein Paket das ACK nicht innerhalb eines bestimmten Zeitraums ankommen, wird das Paket erneut versendet. Das Intervall der wiederholten Versendung wird bei jedem Fehlversuch erhöht, um eine Netzüberlastung zu verhindern.

- Reihenfolgeerhaltung: Bei zuverlässig versendeten Paketen sorgt ENet für den Erhalt der Sendereihenfolge, indem alle Pakete mit höherer Sequenznummmer zurückgehalten werden, bis das Paket mit der erwarteten Sequenznummer empfangen wurde. Unzuverlässig versendete Pakete werden sofort übergeben, was die Latenz auf ein Minimum reduziert. Ein erhaltenes Paket, das eine niedrigere Sequenznummer als ein vorher übergebenes Paket hat, wird verworfen.

- Aggregation: Um die Verbindung optimal auszunutzen, werden alle Protokollkommandos wie z. B. ACK zusammen mit Nutzdaten versendet („piggy-back") und kleinere Einheiten von Nutzdaten in größere Protokollpakete zusammengefasst.

- Kanäle: Um zu verhindern, dass beim Warten auf als *reliable* markierte Daten die als *unreliable* markierten Daten zurückgehalten und verzögert werden, verwaltet die

[4] http://www.cubeengine.com/

Verbindung jeweils getrennte Kanäle für diese beiden Daten-Kategorien, wobei die Sequenznummern unabhängig voneinander hochgezählt werden. So blockiert ein verlorenes Paket nicht die Übergabe von Paketen der anderen Kanäle.

- Fragmentierung und Zusammenbau: Nachrichten, die für ein einzelnes Datagramm zu groß sind, werden in mehrere Datagramme aufgeteilt und beim Empfänger wieder zusammengefügt.
- Überlaststeuerung: ENet implementiert verschiedene Techniken, um die mögliche Bandbreite der Leitung zu erkennen und auf eine Leitungsüberlastung zu reagieren.

2.6.2 Wahl der Netzwerktopologie

2.6.2.1 Client/Server - Architektur

In der Client/Server-Architektur verbinden sich alle Spieler zu einem zentralen Spieleserver, über den während eines Spiels der komplette Datenaustausch fließt. Im Hinblick auf die zusätzliche Verantwortung des Servers existieren zwei verschiedene Ansätze, das *Minimal Client*-Prinzip und das des *Autoritativen Servers*.

Minimal Client Prinzip

Im *Minimal Client* - Ansatz wird die Spielwelt komplett und ausschließlich vom Server berechnet. Vereinfacht dargestellt haben die Clients dabei lediglich die Aufgabe, die Interaktionen des lokalen Spielers an den Server zu übermitteln (siehe Abb. 2.10). Der Server berechnet die dadurch entstehenden Veränderungen in der Spielwelt sowie die Aktionen der serverseitigen Objekte, und schickt regelmäßige Zustandsupdates der Spielwelt an die Clients. Anhand dieser Updates rendern die Clients die Spielwelt zum momentanen Zeitpunkt. Dieser Zeitpunkt liegt allerdings t Zeiteinheiten in der Vergangenheit, wobei t der Zeit entspricht, die der Transport des Updates durch das Netz gedauert hat. Um diese Verzögerung auszugleichen, können Teile der Spiellogik auf den Clients ausgelagert werden, damit die unmittelbar folgenden Serverupdates vorausberechnet werden können. Dieser Ansatz namens *Client Side Prediction* wird in Sektion 2.5.1 erläutert.

Autoritativer Server

Hier wird wie im Peer-to-Peer-Ansatz (siehe Abschnitt 2.6.2.2) die Spiellogik komplett auf jedem Client berechnet. Die Interaktionen werden aber nicht direkt an die anderen Spieler, sondern an einen zentralen Server geschickt. Der Server leitet die Informationen dann an die anderen Clients weiter, berechnet aber auch seinerseits die Spielwelt. Der Hauptvorteil ist, dass der Server als zentrale Autorität dienen kann, und eine verbindliche Sicht auf das

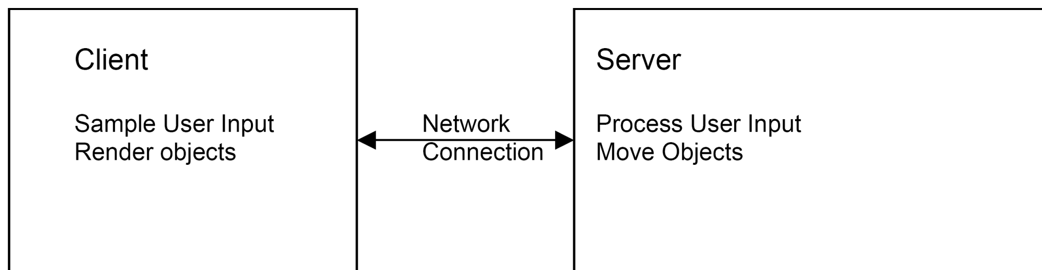


Abbildung 2.10: Minimal Client Prinzip

Spielgeschehen vorgibt. Dadurch können Konflikte und Paradoxien gelöst werden, die durch unterschiedliche Latenzen entstanden sind. Konkrete Fälle hierzu werden in Sektion 2.5.2 beschrieben. Ein weiterer wichtiger Grund in der Praxis ist die Möglichkeit, das Spiel unanfällig gegenüber *Cheating* zu machen. Als Cheating werden im Spielebereich Eingriffe in die Spiellogik bezeichnet, durch die sich ein Spieler auf unfaire Weise Vorteile verschafft. Cheating kann durch Manipulationen am Programmcode mit Hilfe von Reverse Engineering, oder durch parallel gestartete Hilfsprogramme durchgeführt werden, die automatisch Interaktionsdaten generieren oder Datenpakete manipulieren.

Bewertung der Client-Server Architektur:

Vorteile:

- zentrale Referenz macht Synchronisation und Konsistenzerhaltung leichter
- geringer Verwaltungsaufwand für die Clients (jeweils nur eine Netzwerk-Verbindung)
- Anti-Cheat-Technologien können implementiert werden
- problemloser Verbindungsaufbau, da IP und Port allen Clients bekannt sind
- Entlastung der Clients durch serverseitige Berechnungen (z.B. Physik)
- persistente Welten sind nur durch dedizierte Server realisierbar

Nachteile:

- höhere Netzwerklatenz durch Umweg über den Server
- Single Point of Failure
- Server braucht ggf. Wartung

Die Client/Server-Architektur wird heute in fast allen actionlastigen Echtzeitgenres genutzt. Einzige Ausnahme bildet der Bereich der Echtzeit-Strategiespiele, die aufgrund ihrer besonderen Eigenschaften auch in Peer-to-Peer-Architekturen realisiert werden, wie beispielsweise das sehr bekannte *Age of Empires III* (siehe Sektion 2.8.5). Der einzig schwerwiegende Nachteil der Client/Server-Architektur ist der verlängerte Kommunikationsweg. Durch die hohe Verbreitung von Breitband-Internetzugängen hat sich dieser Nachteil aber stark relativiert, da die RTTs zum Server und zurück bei den meisten Spielern heute unterhalb von 50 ms liegen. Bezogen auf den Mobilfunkbereich wirkt dieser Nachteil ebenfalls nicht schwer, da eine Umleitung des Nachrichtenverkehrs über den Neutron-Server keine Verdopplung der RTT bedeutet. Den größten Teil der Gesamtlatenz nimmt das Überwinden der Funkstrecke in Anspruch, während das eigentliche Vermitteln und Weiterleiten der Mobilfunkdaten danach innerhalb weniger Millisekunden über das Internet geschieht. Die zusätzliche Strecke über den Neutron-Server wird also in einem relativ kleinen Teil des Kommunikationsweges zurückgelegt, und verzögert die Gesamtlatenz innerhalb von Europa oder den USA nur um etwa 50-100 ms.

2.6.2.2 Peer-to-Peer Architektur

In der Peer-to-Peer-Architektur kommunizieren die Teilnehmer direkt miteinander, ohne den Umweg über einen Server zu wählen. Jeder Spieler ist dabei mit jedem anderen Spieler verbunden. Der größte Vorteil ist dabei die Minimierung der Kommunikationswege und die damit verbundene Minimierung der Latenz. Allerdings bringt dieser Ansatz auch mehrere Nachteile mit sich. So wird in fast allen Spielegenres das Cheating begünstigt, da Spiellogik komplett auf die Clients ausgelagert wird. Weiterhin erhöht sich der Verwaltungsaufwand für die Clients, da je eine Verbindung zu jedem einzelnen Mitspieler verwaltet werden muss. Dadurch erhöht sich pro Client die Belastung des Prozessors und der Bandbreite, was die Anzahl der Spieler stark begrenzt. Auch ist es kein triviales Problem, einen verteilten Konsistenzalgorithmus zu realisieren, der die durch Latenz entstehenden Inkonsistenzen löst. Einfache Peer-to-Peer Synchronisations-Mechanismen wie die in Abschnitt (2.5.3) vorgestellte *Bucket Synchronisation* kümmern sich nicht um die Konsistenzerhaltung. In der Beurteilung des verteilten Konsistenzalgorithmus „Rendezvous“ (2.5.3) wird der Aufwand deutlich, der betrieben werden muss, um Konsistenz in P2P-Spielen ansatzweise zu realisieren. Alternativ könnte in Anlehnung an das Prinzip des Autoritatven Servers auch ein „autoritativer Peer“ als Referenz dienen. Damit würde der Vorteil der kurzen Kommunikationswege jedoch wieder zunichte gemacht.
Im Bereich der mobilen Netze ergibt sich bei der Peer-to-Peer-Architektur ein zusätzliches Problem, da der Verbindungsaufbau zu den Mitspielern aus folgendem Grund erschwert wird: Wenn im Internet zwei Endgeräte hinter einem NAT-Router betrieben werden, ist es für sie unmöglich sich miteinander zu verbinden, solange sie nicht ihre öffentlichen IPs und Ports kennen, und sie beide gleichzeitig einen Verbindungsaufbau initialisieren. Diese Tech-

nik wird als *NAT Punch-through* [mindcontrol] bezeichnet, und muss je nach Typ der beteiligten Router und Firewalls abgewandelt werden. Da die NAT-Router der Providernetze im Allgemeinen sehr restriktiv sind, sind im Bereich der Mobilfunknetze beim Realisieren von Peer-to-Peer-Architekturen über mehrere Mobilfunkanbieter und -netze hinweg große technische Probleme zu erwarten.

Bewertung der Peer-to-Peer-Architektur

Vorteile:

- Minimale Latenz durch direkte Kommunikationswege
- kein Single Point of Failure

Nachteile:

- cheat-anfällig
- begrenzte Spieleranzahl durch Belastung
- u.U. Verbindungsprobleme durch Firewalls und NAT-Router
- Konsistenzerhaltung ist sehr aufwendig

2.6.3 Minimierung der gefühlten Latenz

Während die tatsächliche Netzwerklatenz schon aufgrund der Ausbreitungsgeschwindigkeit der Daten im physischen Netzwerkmedium niemals unter einen bestimmten Wert fallen kann, so gibt es in Echtzeitspielen dennoch Möglichkeiten, diese Verzögerungen zu kaschieren und teilweise auszugleichen. In vielen verteilten Applikationen und Spielegenres ist es möglich, die visuelle Ausführung einer Interaktion durch Überblend-Effekte u.ä. so in die Länge zu ziehen, dass die Latenz kaum noch wahrgenommen wird [Conner und Holden (1997a)]. Wichtig für ein angenehmes Spielgefühl ist in jedem Fall, alle Interaktionen unverzüglich mit einer visuellen oder auch akustischen Rückmeldung zu quittieren. Wird in Strategiespielen beispielsweise eine Einheit zu einer bestimmten Position geschickt, sollte die Zielposition durch eine animierte Markierung sofort nach dem Klick bzw. Tastendruck angezeigt werden. Dass sich die Einheit erst nach der Dauer einer RTT in Bewegung setzt, wird schon oft deshalb nicht wahrgenommen, weil sie sich außerhalb des sichtbaren Kartenausschnitts befindet. Sogar in 3D-Shootern kann dieses Prinzip teilweise angewendet werden: In Unreal Tournament (siehe Abschnitt 2.8.3) beispielsweise wird das Mündungsfeuer des Raketenwerfers unmittelbar nach Drücken der Feuertaste angezeigt. Das eigentliche Projektil fliegt jedoch erst los, nachdem der Server die Positionsdaten der abgefeuerten Rakete nach einer

vollen RTT übermittelt hat.
Der Ansatz der Kaschierung stößt jedoch an seine Grenzen, sobald eine Entität direkt durch Tastendruck oder durch Mausbewegungen gesteuert wird. Wir haben in Sektion 2.5.1 mit dem Prinzip der Client Side Prediction den Ansatz vorgestellt, lokale Interaktionen sofort umzusetzen, ohne eine Antwort des Servers oder der beteiligten Peers abzuwarten. Dieser Ansatz geht zwangsläufig mit inkonsistenten Sichten auf die Spielwelt einher. Da in actionorientierten Echtzeitspielen fast alle Interaktionen eine Positionsveränderung der Spielfiguren nach sich ziehen, ist hierbei eine zentrale Fragestellung, nach welchem Modell die Positionsdaten ausgetauscht und aktualisiert werden sollen, um die entstehenden Inkonsistenzen möglichst gering zu halten oder möglichst gut zu verbergen. Mit diesen Techniken beschäftigen wir uns im folgenden Unterkapitel.

2.7 Darstellungsmodelle

In einer idealen Umgebung mit uneingeschränkter Bandbreite, uneingeschränkter Prozessorleistung und vernachlässigbarer Latenz wäre es möglich, die Zustandsaktualisierungen entfernter Spielentitäten in der gleichen Frequenz zu senden, in der der Empfänger die Darstellung der Spielwelt aktualisiert. Das 1993 veröffentlichte Spiel *Doom* [idsoftware (1997)] verfolgte diesen Ansatz, und war deshalb nur im lokalen Netz mit maximal vier Spielern gleichzeitig spielbar. Bei der Realisierung von Echtzeitspielen in Netzen mit nicht vernachlässigbarer Latenz und eingeschränkter Bandbreite müssen in der Praxis hingegen folgende Fragen geklärt werden:

1. welche Zustandsinformationen tauschen die Teilnehmer untereinander aus ?
2. wann schickt ein Sender Zustandsupdates eines Spielobjekts ?
3. wo werden entfernte Spielobjekte im aktuellen Frame dargestellt, wenn kein aktuelles Zustandsupdate vorliegt ?

Die beiden grundsätzlichen Prinzipien zur Darstellung entfernter Objekte sind *Extrapolation* und *Interpolation*. Extrapolationstechniken basieren auf dem Prinzip, von vergangenen Zustandsangaben eines Objekts auf dessen aktuellen Zustand zu schließen. Diese Techniken werden allgemein unter dem Begriff $DeadReckoning$ zusammengefasst. Interpolation basiert auf der Übertragung reiner Positionsangaben. Diese Technik wird in Sektion 2.7.3 vorgestellt.

2.7.1 Dead Reckoning

Dead Reckoning wurde im Rahmen der DVEs erstmals in DIS angewandt, und wurde seitdem in vielen Arbeiten weiterentwickelt und variiert. Der Begriff *Dead Reckoning* stammt aus

der Seefahrt, wo diese Technik benutzt wurde, um die Position eines Schiffes auf See zu berechnen. Normalerweise orientierte man sich dabei am Sternenhimmel oder an Navigationsgeräten. Falls das aufgrund von schlechtem Wetter oder Fehlfunktionen nicht möglich war, wurde die aktuelle Position anhand der zuletzt bekannten Position und den aufgezeichneten Kurswechseln und Geschwindigkeiten berechnet.
Das generelle Prinzip bei Dead Reckoning im Bereich der DVEs ist es, Netzwerkverkehr durch lokale Berechnungen zu ersetzen, wann immer man von dem zuletzt bekannten Zustand eines Spielobjekts auf den aktuellen Zustand schließen kann. Wenn beispielsweise bekannt ist, dass es sich bei einem Spielobjekt um ein Projektil handelt, sind nur sehr wenige zusätzliche Informationen nötig, um dessen komplette Flugbahn zu berechnen.
Um Dead Reckoning zu implementieren, muss jeder lokale Teilnehmer das Bewegungsmodell aller entfernten Spielobjekte kennen. Dieses Modell wird *Dead Reckoning Model* (DRM) genannt. Bei jedem Gameloop-Durchlauf wird die aktuelle Position der entfernten Spielobjekte anhand ihrer DRMs berechnet. Sobald ein Spieler die Fahrt- bzw. Laufrichtung ändert, muss diese Richtungsänderung den entfernten Teilnehmern mitgeteilt werden. Normalerweise gibt es einen Toleranzbereich, sodass nicht jeder Tastendruck übertragen werden muss, sondern nur Richtungsänderungen ab einem bestimmten Schwellwert („Threshold“). Um die Größe dieser Abweichung berechnen zu können, verwaltet jeder lokale Simulator auch ein DRM für seine eigene Spielfigur. Mit Hilfe des eigenen DRMs kann eine Entität beurteilen, an welcher Position sie sich in der Darstellung der entfernten Teilnehmer in etwa[5] befindet. Diese lokal simulierte Entität wird auch „Ghost“ genannt. Übersteigt der Abstand der lokalen Entität zum Ghost den festgelegten Schwellwert, wird eine Update-Nachricht an die entfernten Teilnehmer geschickt. Im militärischen Bereich (DIS) enthält ein solches Update u. a. die aktuelle Position, Geschwindigkeit und Ausrichtung, sowie den Zustand von zusätzlichen Subkomponenten (z. B. die Ausrichtung des Geschützes).
Ein weiterer Vorteil von Dead Reckoning ist die Fehlertoleranz gegenüber verloren gegangenen Paketen. Bei Paketverlust kann einfach weiterhin der zuletzt bekannte Zustand als Extrapolationsgrundlage genutzt werden, ohne das Paket neu anfordern zu müssen. Sobald aktuelle Pakete eintreffen, werden die alten Daten hinfällig. Bei kurzen Ausfällen und hoher Updaterate ist die Wahrscheinlichkeit hoch, dass der Paketverlust nicht wahrgenommen wird. In *SIMNET* wurden die Positionsangaben der Entitäten in einer Rate von 15 Updates pro Sekunde aktualisiert, für Flugzeuge lag sie etwas höher [Miller und Thorpe (1995)].

2.7.1.1 Dead Reckoning Modelle

Innerhalb des *Distributed Interactive Simulations*-Standards wurden drei unterschiedlich komplexe Dead Reckoning Modelle (DRMs) definiert, die im Folgenden erläutert werden.
Es sei $P(t)$ die tatsächliche Position einer entfernten Entität zum Zeitpunkt t. Positionsupda-

[5] nur annäherungsweise, da die Latenz der Teilnehmer nicht berücksichtigt wird

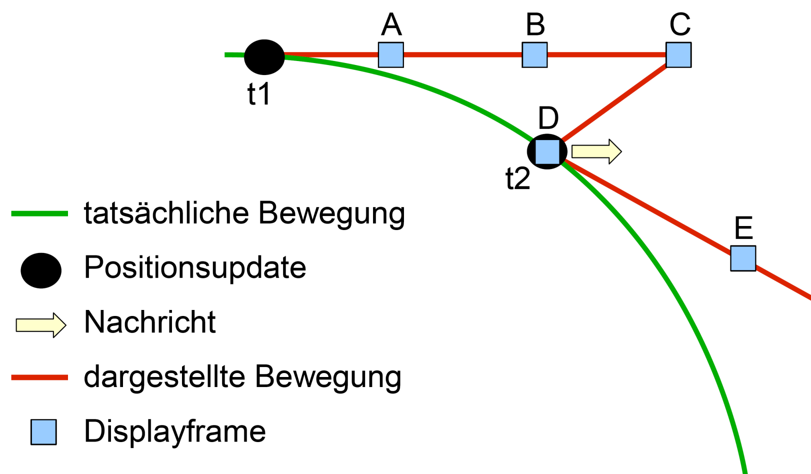


Abbildung 2.11: einfaches Dead Reckoning ohne Konvergenz und Time Compression

tes werden zum Zeitpunkt $t_1, t_2, ...$ empfangen, die wiederum $P(t_1), P(t_2), ...$ beinhalten. Das Dead-Reckoning-Modell gibt eine berechnete Position $D(t)$ für den Zeitpunkt t zurück. Sei t_i das jüngste Update zeitlich vor dem Zeitpunkt t, und $\triangle t$ die Differenz $t - t_i$. Drei DRMs sind denkbar, die sich in der Komplexität der übermittelten Informationen unterscheiden.

1. DRM 0. Ordnung: $D(t) = P(t_i)$
   Es werden keine zusätzlichen Informationen übermittelt und verwaltet, deshalb kann nur die letzte Position zurückgegeben werden.

2. DRM 1. Ordnung: $D(t) = P(t_i) + v(t_i)\triangle t$
   $v(t_i)$ ist der Geschwindigkeitsvektor, der für die entfernte Entität zum Zeitpunkt t_i übermittelt wurde.

3. DRM 2. Ordnung: $D(t) = P(t_i) + v(t_i)\triangle t + (\frac{1}{2})a(t_i)(\triangle t)^2$
   $v(t_i)$ ist der Geschwindigkeitsvektor und $a(t_i)$ ist der Beschleunigungsvektor, der für die Entität zum Zeitpunkt t_i übermittelt wurde. Die Ausrichtung einer entfernten Entität kann auf Basis von periodischen Updates zur Winkelgeschwindigkeit und Winkelbeschleunigung extrapoliert werden.

Welches dieser Modelle am besten geeignet ist, hängt von den Bewegungseigenschaften der modellierten Entität ab. Ein Ziel des praktischen Teils dieser Arbeit wird es sein, eine Beurteilung zur Tauglichkeit der verschiedenen DRMs im Hinblick auf verschiedene Spielgenres und Bewegungsmodelle zu erstellen. Es ist zu erwarten, dass Bewegungsmodelle, die ruckartige Richtungswechsel der Spielfigur erlauben, nicht gut mit einem DRM 2. Ordnung abgebildet werden können, da temporäre Beschleunigungen jeweils nur für einen sehr kurzen Moment gelten. Eine Extrapolation auf Basis dieser Werte wird bei relativ niedrigen

Aktualisierungsraten dementsprechend zu einer hohen Fehlerquote führen. Um den Extrapolationsfehler gering zu halten, werden teilweise auch Hybridmodelle aus verschiedenen DRMs genutzt, wobei diese in Abhängigkeit des momentanen Bewegungsmodus' des modellierten Objekts dynamisch gewechselt werden. Ein Beispiel dafür ist der Egoshooter *Quake* [idSoftware (1999)], dessen Hybridmodell in Sektion 2.8.1 beschrieben wird.
Als weitere Maßnahme zur Einschränkung des Extrapolationsfehlers können Informationen über spezielle Eigenschaften des modellierten Objekts genutzt werden. So kann beispielsweise ein Flugzeug im horizontalen Flug nicht über seine Höchstgeschwindigkeit hinaus beschleunigt werden, und Geschütze sind oft nur innerhalb eines bestimmten Winkels drehbar.

2.7.1.2 Time Compensation

Der bisher betrachtete Ansatz berücksichtigt nicht die Verzögerung, der ein Update beim Versenden über das Netzwerk unterliegt. Sobald eine Positionsangabe bei einem Teilnehmer ankommt, ist sie schon veraltet. Sie zeigt nur an, wo sich die Entität vor L Zeiteinheiten befunden hat, wobei L die Zeit angibt, die für das Versenden und Bearbeiten der Nachricht benötigt wurde. Um die Latenz zu berücksichtigen, kann man die Updatenachrichten mit Zeitstempeln versehen, die in das DRM mit einfließen [Aggarwal u. a. (2004)]. Statt die Positionsangabe eines Updates als einzige Grundlage für die Darstellung im nächsten Displayframe zu nutzen, wird der Zeitstempel der Updatenachricht und die neuen Zustandsinformationen benutzt, um im nächsten Frame die tatsächliche Position zu berechnen. Die Position wird durch $\triangle t$ berechnet, was $(t_d - t_2)$ entspricht, wobei t_d dem Zeitpunkt von Displayframe D entspricht (siehe Abb. 2.12).

2.7.1.3 Konvergenzalgorithmen

Ein zweites Problem bei der bisherigen Herangehensweise sind Sprünge in der Darstellung, die zwischen einzelnen Displayframes entstehen, sobald neue Zustandsdaten einer entfernten Entität eingehen. Dieser Sprung ist deutlich sichtbar in Abb. 2.11 beim Übergang von Position C auf D, und ist in Abb. 2.12 zwar etwas abgeschwächt, aber immer noch vorhanden. Da solche Sprünge unnatürlich und irritierend wirken, ist es wünschenswert, die Zustandsübergänge zu glätten.

Lineare Konvergenz

Der Zustand der Entität soll dazu innerhalb eines kurzen Zeitraums („Konvergenzzeit“) fließend auf die neuen Werte korrigiert werden, und nicht so schlagartig wie bisher. Die grundlegende Herangehensweise bei der *linearen Konvergenz* wird in Abb. 2.13 gezeigt. Sobald eine neue Updatenachricht eintrifft, dient sie dem DRM zunächst als Grundlage zur Berechnung der Position zu einem Zeitpunkt, zu dem wahrscheinlich das nächste Updates eintref-

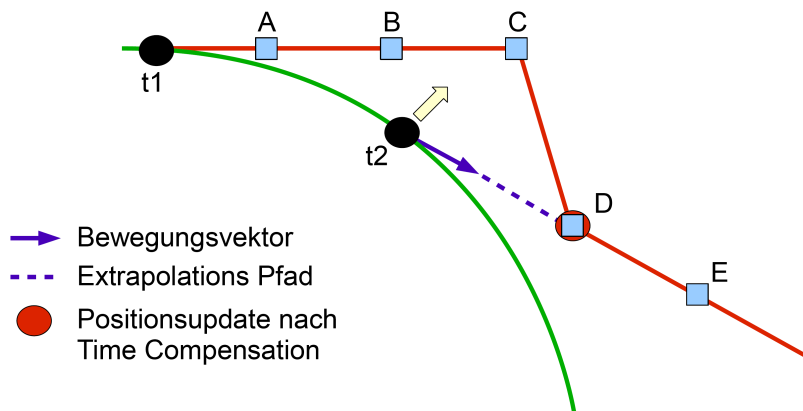


Abbildung 2.12: DR mit Time Compensation

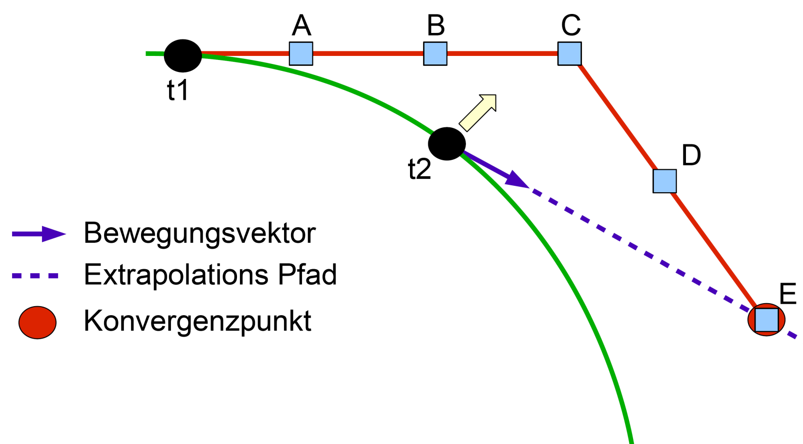


Abbildung 2.13: DR mit linearer Konvergenz

fen wird. Die Position, an der die Entität im nächsten Displayframe angezeigt wird, ergibt sich durch die Interpolation des soeben extrapolierten Wertes mit der zuletzt bekannten Position. Weitere Varianten dieser Technik ergeben sich, wenn man als Interpolationsreferenz nicht die Position des nächsten Updates extrapoliert, sondern des übernächsten, bzw. des K-ten. In unserem ersten Ansatz in Abbildung 2.13 entspricht K dem Wert 1. Größere Werte von K führen möglicherweise (abhängig von den Bewegungseigenschaften der modellierten Entität) zu einem weicheren Übergang zur korrigierten Position, aber bringen auch eine größere durchschnittliche Ungenauigkeit relativ zur tatsächlichen Position mit sich.

Snap-Konvergenz

Die weichere Bewegung, die durch Konvergenzalgorithmen erzwungen wird, wird durch eine größere Abweichung der dargestellten Position zur tatsächlichen Position erkauft. Wie sich

solche Fehler auswirken können, ist in Abbildung 2.14 dargestellt. In diesem Beispiel wird die Position eines Objekts falsch vorausberechnet. Die Anwendung eines Konvergenzalgorithmus' würde in diesem Beispiel dazu führen, dass das Objekt durch eine Wand bewegt wird. Wie gut sich ein Konvergenzalgorithmus für ein bestimmtes Spiel eignet, hängt also nicht nur von der modellierten Entität, sondern auch von der Spielumgebung ab. Insbesondere bei der serverseitigen Trefferauswertung können zusätzliche clientseitige Ungenauigkeiten dazu führen, dass ein Schuss sein Ziel verfehlt, obwohl das Ziel auf dem Bildschirm des Schützen genau im Fadenkreuz war. Unter dieser Problematik litt beispielsweise der Shooter *Quake*, dessen Netcode in Sektion 2.8.1 genauer vorgestellt wird. Es kann deshalb unter Umständen wünschenswert sein, den Fehler möglichst gering zu halten, indem ein Spielobjekt sofort auf die neue Position verschoben wird, sobald ein Update eintrifft. Diese Konvergenz-Variante (bzw. eher das Fehlen jeglicher Konvergenz) wird auch als *Snap-Konvergenz* bezeichnet. In der Praxis kann man sich aber auch unter Nutzung der linearen Konvergenz beliebig nah der Snap-Konvergenz annähern, indem die Konvergenzzeit extrem kurz einstellt wird.

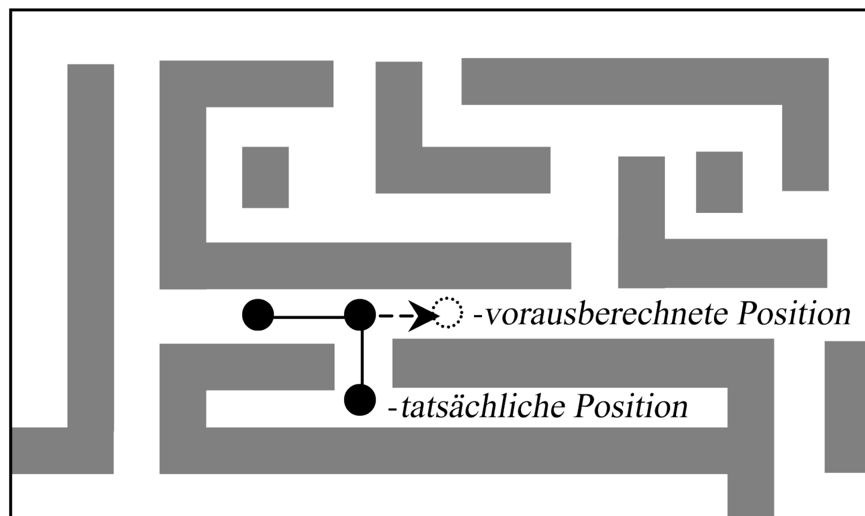


Abbildung 2.14: Fehler bei Extrapolation und Konvergenz

2.7.1.4 Wahl des Schwellwerts

Der Schwellwert, ab dem ein neues Zustandsupdate an die anderen Teilnehmer gesendet wird, ist in Dead Reckoning ein zentraler Parameter im Hinblick auf Genauigkeit und Datenaufkommen. Der ursprüngliche DR-Algorithmus in DIS sieht einen festen Schwellwert vor. Weiterführende DR-Varianten variieren den Schwellwert in Abhängigkeit des Abstands zwischen zwei Entitäten, um Datenaufkommen einzusparen [Cai u. a. (1999)]. Die Idee dabei ist, die Updaterate zwischen zwei Teilnehmern zu senken, falls deren Entitäten außerhalb

eines festgelegten Radius („sensitive region“, (SR)) liegen. Außerhalb der SR ist ein niedriger Schwellwert ausreichend, da zum einen unnatürliche Bewegungen, die durch Konvergenzalgorithmen entstehen, aufgrund der Distanz weniger deutlich wahrgenommen werden. Zum anderen können außerhalb der SR Kollisionen ausgeschlossen werden, wodurch eine wesentlich höhere Toleranz gegenüber abweichenden Positionsdarstellungen gegeben ist. Durch diese Technik kann die Netzwerklast erheblich reduziert werden, da sich normalerweise nur wenige Objekte in unmittelbarer Nähe des lokalen Spielers befinden.

2.7.1.5 Pre-Reckoning

Um zu entscheiden, ob im aktuellen Frame ein Update generiert werden muss, betrachtet der DIS-DR Algorithmus nur die Größe des Abstands der Ghost-Entität zur tatsächlichen Position. Ein Update wird also erst generiert, wenn der Schwellwert bereits überschritten ist. Eine weiterer Schritt zur Einschränkung von Extrapolationsfehlern ist es, Updates schon in Situationen zu senden, die auf ein baldiges Überschreiten des Thresholds hindeuten. In der in [Duncan und Gracanin (2003)] vorgestellten Variante namens *Pre-Reckoning* werden dazu Geschwindigkeits- und Richtungswechsel berücksichtigt. Ein baldiges Überschreiten des Thresholds steht demnach in drei verschiedenen Situationen bevor:

1. ein stehendes Objekt beginnt sich zu bewegen
2. ein sich bewegendes Objekt hält an
3. ein Objekt macht einen scharfen Richtungswechsel

Die ersten beiden Fälle können entweder durch ein Vergleichen des Betrags der Geschwindigkeitsvektoren erkannt werden, oder anhand der Benutzerinteraktionen (Vorwärts-Taste gedrückt bzw. losgelassen). Der dritte Fall kann erkannt werden, indem der Winkel der beiden Geschwindigkeitsvektoren betrachtet wird. Diesbezüglich wird ein weiterer Schwellwert definiert, der die Größe des Winkels beschreibt, ab dessen Überschreiten ein Update generiert und versendet wird. Sind bei der modellierten Entität abrupte Richtungswechsel möglich (wofür Pre-Reckoning auch konzipiert wurde), sollte der Schwellwert auf etwa 80° gesetzt werden.

Bewertung:
Während DIS-DR auf die Modellierung von relativ trägen Fahr- und Flugzeugen abzielt, wurde Pre-Reckoning für Objekte konzipiert, bei denen sich Geschwindigkeit und Bewegungsrichtung schnell ändern können. Es eignet sich deshalb unter anderem für die Anwendung in actionlastigen Spielen mit menschlichen Avataren. Der Preis für den erhöhten Grad an Konsistenz sind eine geringfügig erhöhte Prozessorlast und ein etwas erhöhter Netzwerkverkehr, da bei niedrigen Schwellwerten in manchen Situationen auch

Updates generiert werden, auf die keine Überschreitung des Positions-Schwellwerts erfolgt wäre. Wird ein sehr kleiner Schwellwert gewählt, relativiert sich dieser Vorteil allerdings. Der grundsätzliche Ansatz, das Senden von Updates nicht (nur) vom Abstand des Ghosts abhängig zu machen, sondern (auch) von Interaktionen, ist jedoch sehr hilfreich im Hinblick auf die Steigerung der Konsistenz. Ob sich dieser Ansatz für ein Spiel lohnt oder nicht, muss von Fall zu Fall entschieden werden. Im Falle von Spielen mit direkter Steuerung ist Pre-Reckoning quasi automatisch gegeben, da praktisch jeder Tastendruck zu einem Richtungswechsel um mehr als 45° führt, wodurch bei halbwegs vernünftiger Parametrisierung auf jeden Fall der Orientierungs-Schwellwert überschritten und sofort ein Update gesendet wird.

Vorteile:

- Einschränkung des Extrapolationsfehlers

Nachteile:

- zusätzliche Prozessorlast, wenn auch nur sehr gering
- ggf. leicht erhöhtes Transfervolumen durch das Generieren überflüssiger Updates

2.7.2 Position History Based Dead Reckoning

Eine andere DR-Variante verzichtet auf die Übertragung von Geschwindigkeit und Beschleunigung. Stattdessen wird nicht, wie im DIS-DR-Ansatz, nur das letzte Zustandsupdate als Berechnungsgrundlage betrachtet, sondern die Positionen und Orientierungswinkel der letzten zwei oder drei Updates. Die Verarbeitung von Zustandsupdates erfolgt in zwei Schritten (siehe Abb. 2.15):

- Extrapolations-Schritt
  Der Empfänger berechnet die wahrscheinliche Position des nächsten Zustandsupdates der entfernten Entität voraus, indem er eine Kurve durch die letzten drei Positionsangaben legt. Der berechnete Punkt wird „Konvergenzpunkt“ genannt.
- Konvergenz-Schritt
  Es wird eine Konvergenzlinie von der aktuellen Anzeigeposition bis zum errechneten Konvergenzpunkt berechnet, auf der das Objekt in den folgenden Displayframes bewegt wird, um eine flüssige Animation zu gewährleisten.

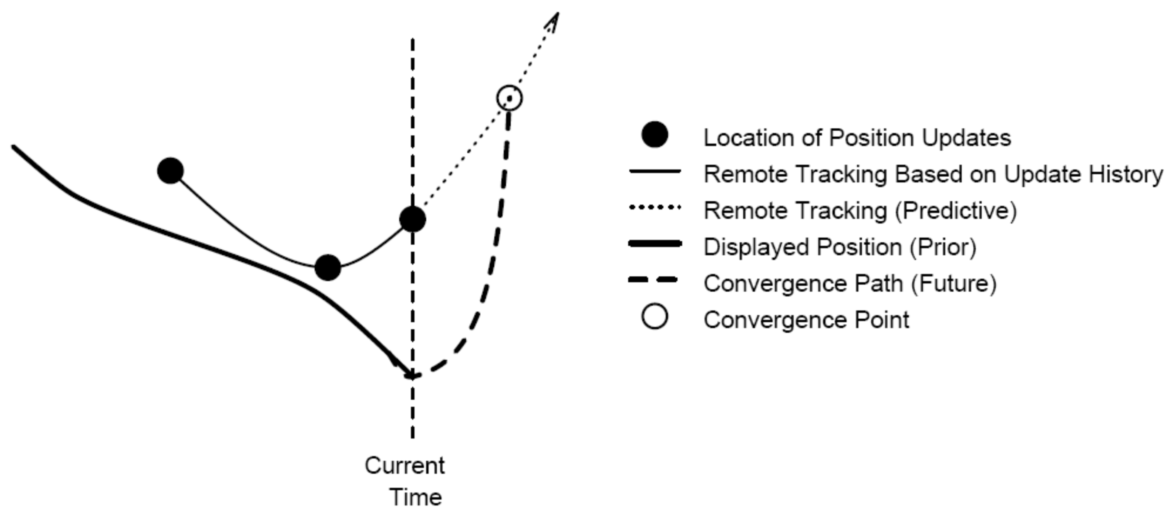


Abbildung 2.15: PHB-DR: Tracking- und Konvergenzschritt [Singhal und Cheriton (1994)]

Adaptiver Extrapolations-Algorithmus

Der Konvergenzpunkt wird entweder durch eine Funktion 2. Grades berechnet, die eine Parabel durch die drei letzten Positionsangaben bildet, oder durch eine lineare Funktion, die eine Gerade durch die letzten beiden Positionsangaben bildet (siehe Abbildung 2.16). Welche der beiden Funktionen gewählt wird, hängt vom Winkel der letzten drei Positionsangaben („angle of embrace") ab. Große Winkel entsprechen einer leichten Bewegung, die sich wahrscheinlich in der gleichen sanften Kurve weiterbewegen wird. Deshalb wird in diesem Fall die Parabelfunktion gewählt (vgl. Abb. 2.16 b)). Spitze Winkel implizieren eine abrupte Richtungsänderung, die beispielsweise durch eine Kollision ausgelöst wurde. Hier wird die lineare Funktion gewählt, da eine Weiterbewegung des Objekts in Parabelform unwahrscheinlich ist (Abb. 2.16 a)).

Adaptiver Konvergenz-Algorithmus

Die Kurve, auf der das entfernte Objekt bis zum nächsten Positionsupdate zum Konvergenzpunkt weiterbewegt wird, wird ebenfalls abhängig vom zuvor berechneten „angle of embrace" gewählt. Ein großer Winkel entspricht einer nahezu linearen Bewegung in konstanter Geschwindigkeit. In diesem Fall wird auf die lineare Konvergenzfunktion zurückgegriffen (siehe Abb. 2.17 b)). Ein spitzer Winkel wird als scharfe Kurve interpretiert, die besser durch eine Parabelform beschrieben werden kann (Abb. 2.17 a)).

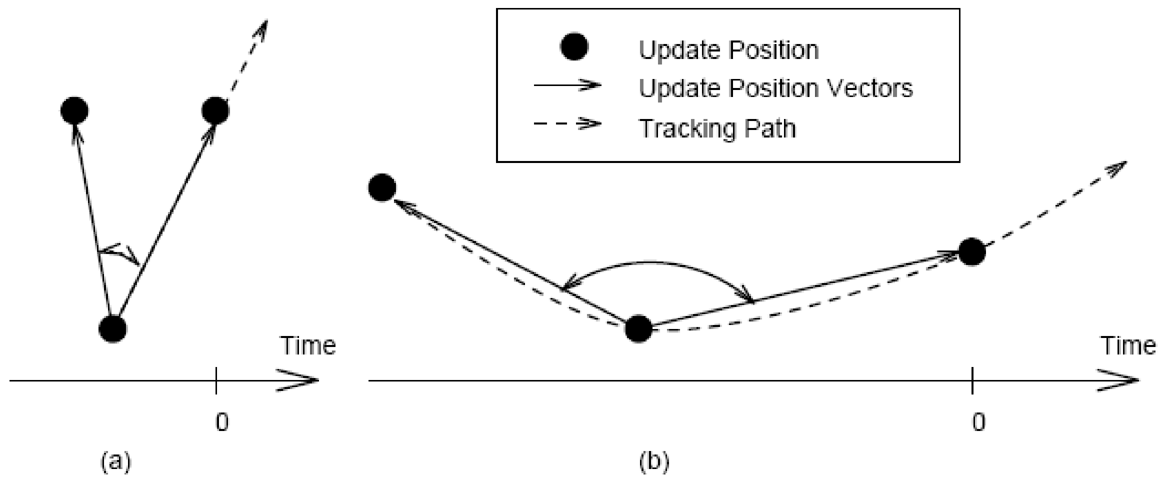


Abbildung 2.16: PHB-DR: Extrapolationsfunktionen [Singhal und Cheriton (1994)]

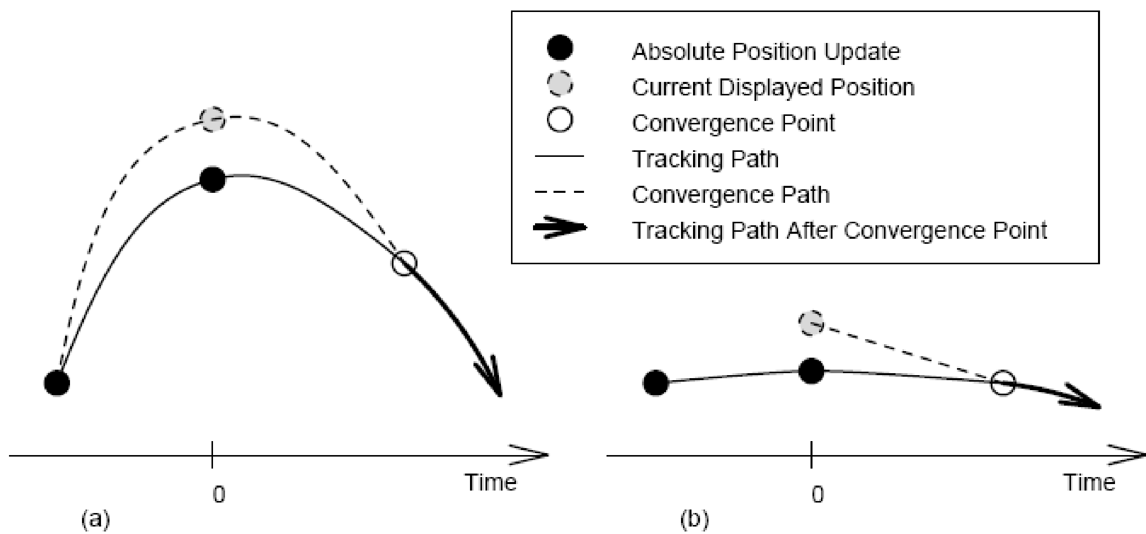


Abbildung 2.17: PHB-DR: Konvergenzfunktionen [Singhal und Cheriton (1994)]

Winkel zwischen den letzten drei Updates	Kurventyp	Extrapolationsmodell	Konvergenzmodell
klein	scharfe Kurve	1.Grades	2. Grades
mittel	weiche Kurve	2. Grades	2. Grades
groß	gerade Linie	2. Grades	1. Grades

Tabelle 2.3: Adaptive Extrapolation und Konvergenz

Bewertung:
Messungen zeigen [Singhal und Cheriton (1994)], dass PHB-DR dem DIS-DR in vielen Anwendungsfällen mindestens ebenbürtig ist. In Umgebungen mit vernachlässigbarer Latenz ist DIS-DR leicht überlegen, da es bei 25% weniger Datenaufkommen eine niedrigere relative Fehlerquote erreicht. Das Einbeziehen von Beschleunigung in DIS führt in Netzwerken mit hoher Latenz aber zu einer Verstärkung des relativen Fehlers, da sich veraltete Beschleunigungsdaten besonders empfindlich auf die momentane Abweichung auswirken. PBH-DR führt insbesondere bei ruckartigen Bewegungen zu einer niedrigeren relativen Fehlerquote, wodurch es sich prinzipiell als DR-Modell für menschliche Akteure in Actionspielen anbietet.

2.7.3 Interpolation

Alle bisher vorgestellten DR-Techniken basieren auf dem Prinzip der Extrapolation. Alle Extrapolationsmodelle setzen voraus, dass sich Geschwindigkeit und Richtung bis zum nächsten Zustandsupdate nicht schlagartig ändern werden, und deshalb als Grundlage für die Berechnung der nächsten Position herangezogen werden können. Dies ist jedoch nicht für alle Spielgenres und Bewegungsmodelle korrekt. In Sport- und Actionspielen, in denen abrupte Richtungswechsel der Spielfigur möglich sind, führen Extrapolationen oft zu hohen Fehlerquoten. Eine einfache Alternative ist deshalb die regelmäßige Übertragung der reinen Positionsangaben eines Objekts ohne Beachtung von Geschwindigkeit und Beschleunigung. Da die Positionsangaben entfernter Objekte zum Zeitpunkt des Eintreffens aufgrund der Latenz schon veraltet sind, bewegt sich das Objekt dabei ständig in der Vergangenheit. Wenn der Sender beispielsweise 10 Positionsupdates pro Sekunde schickt, könnte in der lokalen Darstellung auf Seiten des Empfängers eine Verzögerung („Deltazeit“) von 100 Millisekunden implementiert werden. Während die Frames gerendert werden, wird die Position des Objekts zwischen dem letzten bekannten Update und dem vorletzten Update (oder alternativ: der letzten Render-Position) über diese 100 Millisekunden interpoliert. Kurz bevor das Objekt im letzten Frame an die zuletzt bekannte Position verschoben wird, trifft das nächste Positionsupdate beim Empfänger ein, da 10 Updates pro Sekunde einem Update-Intervall von 100 ms entsprechen.
Falls ein Paket nicht rechtzeitig ankommt, gibt es zwei Möglichkeiten: Entweder greift der Empfänger auf Extrapolation zurück, oder er lässt das Objekt einfach an der zuletzt bekannten Position. Im ersten Fall werden Extrapolationsfehler in Kauf genommen, die zweite Alternative führt zu einer stockenden Bewegung des Objekts. Um gelegentlichen Paketverlust einzukalkulieren, könnte man sich auch für eine Verdopplung der Deltazeit entscheiden, in unserem Beispiel auf 200 ms. Dadurch würde das Objekt trotzdem weiterhin zu einer gültigen Position bewegt werden, auch wenn ein einzelnes Update verloren geht. Die dadurch entstehende flüssigere Darstellung des entfernten Objekts erkauft man sich durch eine zusätzliche Abweichung zur tatsächlichen aktuellen Position.

Allgemeiner Interpolationsalgorithmus:

Es wird hierbei vorausgesetzt, dass sich die Empfängeruhr der Uhr des Senders anpasst, ohne Berücksichtigung der Netzwerklatenz. Der Sender schickt dem Empfänger also seine Uhrzeit, und der Empfänger stellt seine eigene Uhrzeit blind darauf ein. Die Uhren von Sender und Empfänger werden also immer synchronisiert sein, wobei die Empfängerzeit etwas nachgeht. Diese Abweichung entspricht der Latenz zwischen Sender und Empfänger.

1. jedes Update enthält den Sender-Zeitstempel mit dem Zeitpunkt der Erstellung

2. der Empfänger berechnet in jedem Gameloop-Durchlauf eine Zielzeit T_{Ziel}, indem er die definierte Deltazeit (100 ms) von seiner aktuellen Uhrzeit abzieht.

3. Falls T_{Ziel} zwischen dem Zeitstempel des letzten Updates und dem des vorletzten Updates liegt, dann ergibt sich aus diesen Zeitstempeln, welcher Teil des Update-Intervalls im aktuellen Frame vergangen ist (ΔI)

4. ΔI wird benutzt, um Position und Ausrichtungswinkel des Objekts im aktuellen Frame zu interpolieren

Prinzipiell wird bei der Interpolation also empfängerseitig ein zusätzlicher Zeitpuffer eingerichtet, der in unserem Beispiel 100 ms beträgt. Alle entfernten Spieler werden somit an einer Position dargestellt, an der sie sich zu einem Zeitpunkt in der Vergangenheit befanden. Dieser Zeitpunkt entspricht der Latenz des Empfängers, plus der $Deltazeit$, über die interpoliert wird.

Interpolationsfehler

Die Verwendung von Interpolation setzt das Generieren und Versenden von Updates in einer festen Rate voraus. Da die Updaterate in der Praxis durch die Leistungsfähigkeit der Endgeräte und die Bandbreite des Netzwerks beschränkt wird, ergeben sich dadurch klassische Samplingfehler, wie sie auch beim Digitalisieren analoger Signale auftreten. Ist das modellierte Objekt beispielsweise ein hüpfender Ball, so befindet sich das Objekt bei den Extremwerten entweder am Scheitelpunkt hoch in der Luft, oder es trifft gerade auf den Boden. Die meiste Zeit jedoch wird sich der Ball irgendwo dazwischen befinden. Wann immer zur nächsten empfangenen Position interpoliert wird, ist es sehr unwahrscheinlich, dass diese Position genau dann gesendet wurde, als sich das Objekt gerade an einem Scheitelpunkt

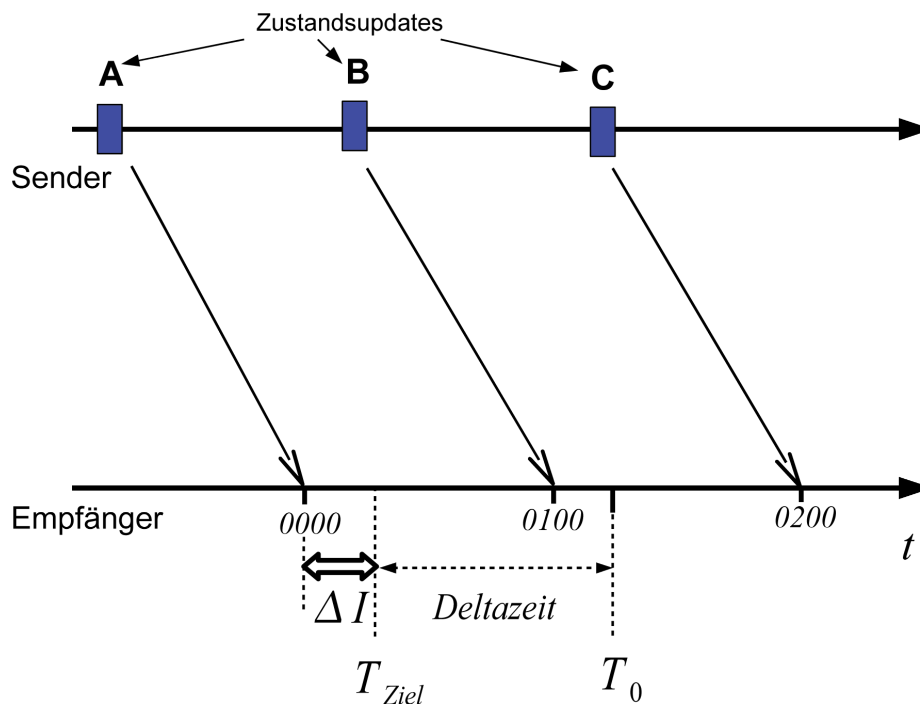


Abbildung 2.18: Interpolation

befand. Als Folge wird die Bewegung beim Empfänger abgeflacht dargestellt (vgl. Abbildung 2.19).

Interpolation mit Position History

Dieses Problem wird noch verstärkt, wenn die Deltazeit so hoch gewählt ist, dass T_{Ziel} mehrere Updates in der Vergangenheit liegt. Da im bisher betrachteten Interpolationsalgorithmus als zweiter Referenzpunkt immer die Position aus dem aktuellsten Update gewählt wird, werden alle weiteren Updates ignoriert, die zwischen T_{Ziel} und dem aktuellsten Update empfangen wurden. Dadurch folgt die lokale Darstellung nicht dem ursprünglichen Bewegungsverlauf, sondern wählt sozusagen die direkte Abkürzung zur aktuellsten Position. Um das dadurch entstehende zusätzliche Abflachen des Bewegungsverlaufs zu verhindern, benutzt der 3D-Shooter *Half-Life* [Bernier (2001)] folgende Technik:

Alle empfangenen Updates werden zusammen mit dem Zeitstempel des Empfangszeitpunktes in einer Datenstruktur namens $PositionHistory$ abgelegt. Die Zielzeit T_{Ziel} wird wie bisher berechnet. Anhand der Zielzeit und der Zeitstempel wird die $PositionHistory$ nach den beiden Updates durchsucht, die zeitlich direkt vor und direkt hinter T_{Ziel} liegen. Position und Ausrichtungswinkel des Objekts im aktuellen Frame wird anhand dieser beiden Updates

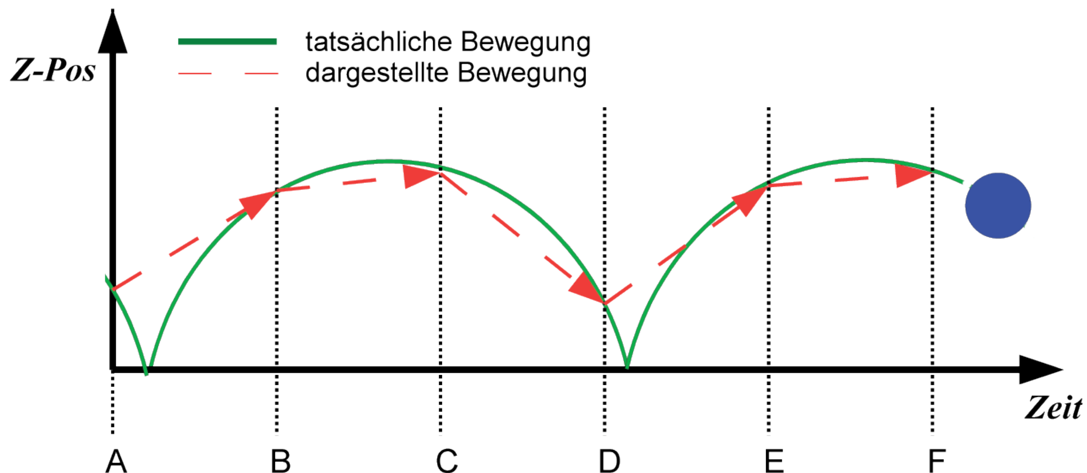


Abbildung 2.19: Interpolationsfehler

und ihrer Zeitstempel interpoliert.

Bewertung:
Durch die Verwendung einer Position History wird der Bewegungsverlauf nur so gut beschrieben, wie es die Updaterate zulässt. Das Anlegen, Verwalten und Durchsuchen der Datenstruktur verursacht nur geringe Speicher- und Prozessorlast, da die Deltazeit normalerweise nur sehr wenige Updates umfassen wird, um die Inkonsistenzen nicht unnötig zu vergrößern. Die Position History wird also immer nur wenige Einträge beinhalten, zumal Einträge sofort gelöscht werden können, sobald ihr Zeitstempel unterhalb von T_{Ziel} liegt.
Interpolation im Allgemeinen sorgt bei ausreichend hoher Updatefrequenz Deltazeit und für einen äußerst flüssigen und originalgetreuen Bewegungsverlauf. Aufgrund des hohen Positionsfehlers eignet es sich jedoch für nur sehr wenige Spieletypen. Prominentester Vertreter ist der teambasierte Ego-Shooter Counter-Strike, der auf der Half-Life-Engine basiert (siehe Sektion 2.8.2).

Vorteile:

- keine Prediction-Fehler
- äußerst saubere Wiedergabe des Bewegungsverlaufs

Nachteile:

- konstante Updaterate in ausreichend hoher Frequenz nötig

- relativ hohes Datenaufkommen auch bei geringer Interaktion
- zusätzliche Verzögerung verstärkt die Abweichung zur tatsächlichen aktuellen Position
- empfindlich gegenüber Paketverlust
- Abflachen das Bewegungsverlaufs bei ruckartigen Richtungswechseln

2.8 Repräsentative Beispiele aus der Welt der PC-Spiele

2.8.1 First Person Shooter: Quake

Quake wurde am 22. Juni 1996 vom US-amerikanischen Softwarehaus idSoftware veröffentlicht. Quake ist einer der ersten Multiplayer FPS, die als Netzwerkprotokoll TCP/IP unterstützten, und war damit grundsätzlich auch über das Internet spielbar.
Wie alle gängigen Ego-Shooter basiert Quake auf einer Client/Server Architektur. Das Hauptargument bei der Wahl dieser Architektur ist das Erschweren des clientseitigen Mogelns („Cheating“) durch Eingriff in die Spiellogik mit Hilfe von Disassemblern oder durch Manipulation der Datenpakete. Die Hauptschleifen von Client und Server sind wie folgt aufgebaut:

Client-Loop

1. Startzeit speichern
2. User Eingaben (Maus, Tasten, Joystick) abfragen und speichern
3. Interaktionskommandos zusammenfassen und abschicken, zusammen mit der Simulationszeit (Dauer des letzten clientseitigen Gameloop-Durchlaufs)
4. Serverpakete auslesen
5. Zustand der sichtbaren Objekte anhand der Serverpakete aktualisieren
6. aktuelle Szene rendern
7. Endzeit speichern
8. Endzeit − Startzeit = Simulationszeit des nächsten Frames

Server-Loop

1. Startzeit speichern
2. Userkommandos aus den Datenpaketen der Clients auslesen
3. Userinteraktionen ausführen
4. servergesteuerte Spielobjekte anhand der Simulationszeit (Dauer des letzten serverseitigen Schleifendurchgangs) berechnen
5. Endzeit speichern
6. Endzeit − Startzeit = Simulationszeit des nächsten Frames

NetQuake

Der Netcode dieser ersten Version, die später als „NetQuake“ bezeichnet wurde, entsprach dem Minimal- Client-Prinzip (siehe 2.6.2.1). Der Server empfängt und evaluiert dabei ständig die Kommandos der teilnehmenden Spieler, wie z.B. "Vorwärtstaste zum Zeitpunkt t für eine Dauer von 50ms gedrückt". Anhand dieser Kommandos errechnet der Server die neue Spielsituation, wie z. B. die aktuelle Position aller Spielobjekte. Jeder Client bekommt in einer konstanten Rate die Positionen und Zustände aller für ihn relevanten Spielentitäten übermittelt, so auch die der eigenen Spielfigur. Diese Server-to-Client Pakete enthalten auch Informationen darüber, welches Kommando des lokalen Spielers zuletzt empfangen und berücksichtigt wurde.
Der Vorteil dieses Ansatzes ist, dass die Konsistenz der Spielwelt bei allen Teilnehmern durch den Server als oberste Autorität erzwungen wird. Der Nachteil ist, dass sämtliche Aktionen um die jeweilige Latenz verzögert werden (siehe Abschnitt „Interaktivität“ in 2.4.2). Das ist im lokalen Netzwerk bei Latenzen unter 10 ms akzeptabel, aber für Spieler mit Dial-up Modems und Latenzen von ca. 100 ms wurde das Spiel dadurch unspielbar. Oft hatte man in Quake das Gefühl, wie auf Eis zu gleiten, bis endlich die jeweils letzte Richtungsänderung vom Server bestätigt wurde. Nicht nur das Spielgefühl litt, sondern auch die Gewinnchancen. Spieler mit niedrigem Ping (in der Spielerszene gerne als „Low Ping Bastards“ oder kurz: „LPBs“ tituliert) konnten exakter manövrieren und schneller reagieren als Spieler mit hohem Ping ("High Ping Bait"/ "HPBs").
idSoftware überarbeitete den Netzwerkcode und baute nachträglich die bereits in Sektion 2.5.1 vorgestellte Technik *Client Side Prediction* ein. Der überarbeitete Quake-Build wurde unter dem Namen *QuakeWorld* am 17.Dezember 1996 als kostenloser Patch veröffentlicht. Der Source Code von Quake und QuakeWorld wurde im Jahr 1999 unter der Gnu Public License freigegeben [idSoftware (1999)].

Client Side Prediction in Quake

Zur Realisierung der Client Side Prediction wird in QuakeWorld eine Kommando-Datenstruktur vom Client zum Server übertragen, die folgende Daten enthält:

```
typedef struct usercmd_s
{
    byte msec;        // Dauer des Kommandos in ms (Framedauer)
    vec3_t angles;    // Blickrichtung
    short forwardmove, sidemove, upmove;
     // Steuerimpulse von Maus/Joystick
    byte buttons;     // Byte-Array der gedrückten Tasten
} usercmd_t;
```

Diese Datenstruktur wird vom Client in jedem lokalen Gameloop-Durchgang erstellt. Der Client schickt diese Datenstrukturen in einer wählbaren konstanten Rate an den Server, wobei ggf. mehrere Kommandos zusammengefasst werden. Der Server berechnet anhand dieser Daten die neue Position der Spielfigur, und schickt (ebenfalls in einer konstanten Rate) Antwortpakete an den Client zurück. Diese Pakete enthalten Zustandsupdates aller Spielentitäten, die für den Spieler momentan sichtbar sind, inklusive des Zustands der Client-Spielfigur, sowie die laufende Nummer des dabei zuletzt berücksichtigten Kommandos. Die Zustandsdaten beinhalten unter anderem die Position, Geschwindigkeit, und den aktuelle Bewegungsmodus der Entität. Der Bewegungsmodus gibt an, welche Art von Bewegung die Entität momentan ausführt, damit die jeweils aktuelle Position zwischen den Zustandsupdates clientseitig genauer in- bzw. extrapoliert werden kann. Folgende Modelle sind dabei möglich:

Bewegungsmodi:

- TR_STATIONARY, TR_INTERPOLATE
  Dead-Reckoning-Modell 0. Ordnung
  für stationäre Entitäten, oder falls der Client die Extrapolation im Optionsmenü deaktiviert hat

- LINEAR / LINEAR_STOP
  Dead-Reckoning-Modell 1. Ordnung
  Extrapolation für lineare Bewegung

- TR_SINE
  Dead-Reckoning-Modell 1. Ordnung
  Extrapolation für springende Entitäten (Sinus-Funktion)

- TR_GRAVITY
  Dead-Reckoning-Modell 1. Ordnung
  Extrapolation für fallende Entitäten (Z-Position wird zusätzlich verringert)

Im Interpolationsmodus verzichtet Quake auf Extrapolation gemäß der genannten Dead-Reckoning- Algorithmen, und stellt die Spielobjekte gemäß der in Abschnitt 2.7.3 beschriebenen Technik der Interpolation ohne Position History dar. Dadurch wird das Zielen allerdings erheblich erschwert, weil die Positionen der Gegner nur dann der tatsächlichen Position entsprechen, wenn sich der Gegner nicht bewegt. Weil das in einem Quake-Match so gut wie nie vorkommt, ist der Interpolationsmodus standardmäßig deaktiviert.
Im Extrapolationsmodus wird der Gegner an der korrekten Position angezeigt, solange er keine abrupten Richtungswechsel vornimmt. Um durch Richtungswechsel verursachte Extrapolationsfehler einzuschränken, ist in Quake die Time Compensation (siehe 2.7.1.2) auf eine Zeitspanne 100 ms beschränkt. Bei Spielern mit einer Latenz von mehr als 100 ms

werden alle entfernten Spieler dementsprechend an einer falschen Position angezeigt. Je deutlicher die Latenz die 100 ms überschreitet, desto stärker muss der Spieler vorhalten.

Implementierung der Client Side Prediction:

Bis ein entsprechendes Antwortpaket des Servers eingeht, versucht der Client die Position seines lokalen Avatars vorherzusagen, indem er blind von einer Bestätigung des Servers ausgeht. Er benutzt dabei die exakt gleichen Algorithmen, die auch der Server nach Erhalt der Kommandos anwenden wird. Als Startpunkt geht der Client dabei vom letzten Kommando aus, das vom Server bestätigt wurde. Dieses Kommando liegt irgendwo in der Vergangenheit, etwa um die durchschnittliche RTT verzögert. Wenn der Client z. B. die Spielwelt in einer Rate von 50 Frames pro Sekunde rendert (was einer Framedauer von 20 ms entspricht), und eine Latenz von 100 ms RTT zum Server hat, wird der Client 5 Kommandos speichern und ausführen (eins pro Frame), bis er die Antwort zu Kommando Nr. 1 vom Server erhalten hat. Als wichtigstes Ergebnis erhält der Client dadurch die Position, an der sich seine Spielfigur im aktuellen Frame befindet. Der allgemeine Algorithmus der Client Side Prediction sieht wie folgt aus:

```
"startzustand" := Zustand nach dem Kommando, das
                   zuletzt vom Server bestätigt wurde
"kommando" := erstes User-Kommando nach dem zuletzt vom
                   Server bestätigten Kommando

while (true)
{
  führe "kommando" auf "startzustand" aus,
    um "zielzustand" zu erhalten;

  if ("kommando" war das aktuellste User-Kommando)
     break;

  "startzustand" := "zielzustand";
  "kommando" := nächstes Kommando;
};
```

Das Prinzip des *Minimal Clients* wird im Rahmen der *Client Side Prediction* also insofern erweitert, dass ein Teil der Spiellogik ebenfalls auf dem Client läuft, um einen Teil der unmittelbar folgenden Spielsituation vorhersagen zu können. Der Nachteil der Client Side Prediction ist, dass dadurch die Konsistenz zum Spielgeschehen auf dem Server verloren geht. Der

Client hat nach wie vor keinen autoritativen Einfluss auf die Simulation der Spielwelt, wie es bei einem Peer-to-Peer Ansatz der Fall wäre. Stattdessen gibt der Server als oberste Instanz nach wie vor die einzig verbindliche Sicht auf das Spielgeschehen vor. Die Clients versuchen lediglich, sich dieser Sicht möglichst perfekt anzunähern, was aufgrund der Netzwerklatenz nie vollständig gelingen kann. Dadurch ergeben sich folgende Schwierigkeiten:
Da die clientseitig vorherberechnete Simulation regelmäßig durch den Server korrigiert wird, kann es bei großen Abweichungen zu merklichen Sprüngen in der Darstellung der Spielobjekte kommen. Diese Abweichungen können zum einen durch Kollisionen mit anderen Spielobjekten entstehen, die clientseitig nicht vorhergesehen werden konnten. Zum anderen erzwingt Quake im Falle von Paketverlust / zu großer Latenz die Synchronisation mit dem serverseitigen Zustand, falls die Spanne zwischen dem aktuell clientseitig ausgeführten Kommando und dem letzten serverseitig bestätigten Kommando zu groß wird.

Trefferauswertung

Das zweite Problem, das durch die clientseitige Inkonsistenz entsteht, betrifft die Zielproblematik, die schon in Abschnitt 2.5.2 erwähnt wurde. Aufgrund der Fehler, die durch Extra-/Interpolation und Konvergenz entstehen, ergibt sich eine gewisse Abweichung in der clientseitigen Darstellung einer Entität und der tatsächlichen Position aus Sicht des Servers. Ob ein Schuss sein Ziel trifft oder nicht, entscheidet der Server anhand der ihm bekannten Positionsdaten. Schießt ein Spieler mit einer Instant-Hit-Waffe auf ein Ziel, so muss er deshalb seine Latenz einkalkulieren und entsprechend vorhalten. Wenn beispielsweise der lokale Spieler mit einem Ping von 100 ms auf einen Gegner zielt, der exakt rechtwinklig zur seiner Blickrichtung mit einer Geschwindigkeit von 500 Pixeln pro Sekunde läuft, dann muss der lokale Spieler mit einer Instant-Hit Waffe 50 Pixel vor den Gegner zielen, um zu treffen. Je höher die Latenz, desto weiter muss vorgehalten werden. Der Spieler muss also ein Gefühl für seine eigene Latenz entwickeln. In QuakeIII wurde deshalb ein kurzes akustisches Signal als Trefferanzeige eingebaut. Bei hoher Latenz, Jitter und hektisch ausweichenden Gegnern wird das Zielen trotzdem immer mehr zur Glückssache. Eine elegantere Lösung führte das Softwarehaus Valve im Jahr 1998 mit der Veröffentlichung des 3D-Shooters Half-Life und der darin implementierten Technik namens *Lag Compensation* ein, die in Sektion 2.8.2 erläutert wird.

Trotz der genannten Probleme empfand die Mehrheit der Spieler QuakeWorld als erhebliche Verbesserung, während einige Spieler (LPBs) das ursprüngliche Netzwerkmodell bevorzugten.

2.8.2 First Person Shooter: Half-Life / Counter-Strike

Half-Life wurde am 31. Oktober 1998 vom Softwarehaus Valve veröffentlicht. Es basiert auf der QuakeWorld-Engine, die Valve von idSoftware lizensierte. Valve erweiterte den Netcode jedoch um eine Technik namens *Lag Compensation*, die im nächsten Abschnitt erläutert wird. Der Sourcecode von Half-Life wurde bis heute nicht freigegeben, aber Valve veröffentlichte detaillierte Erläuterungen zum erweiterten Netzwerkcode [Bernier (2001)]. Auf Basis der erweiterten Half-Life-Engine wurde der teambasierte Taktik-Shooter *Counter-Strike* entwickelt, der bis heute der meistgespielte Ego-Shooter weltweit ist.

Lag Compensation

Das Problem des in Quake verfolgten Ansatzes besteht darin, dass aufgrund der Latenz ein exaktes Zielen unmöglich ist, wie im Absatz „Trefferauswertung“ in Sektion 2.8.1 erläutert wurde. Die einfachste Lösung dazu bestünde darin, die Trefferauswertung clientseitig zu implementieren, um sich dabei auf die clientseitigen Koordinaten des Zielobjekts beziehen zu können, statt auf die tatsächliche Position aus Sicht des Servers. Der Client müsste dazu Nachrichten versenden, die Informationen beinhalten wie "bin an Position x,y und habe soeben Spieler B abgeschossen". Dieser Ansatz wird in Peer-to-Peer Architekturen und militärischen Simulatoren verfolgt, die von vertrauenswürdigen Clients ausgehen können. In der Welt der kommerziellen PC-Spiele ist dieser Ansatz nicht akzeptabel, weil es auf diese Weise unmöglich ist, Cheating auf ein erträgliches Maß zu reduzieren.
Die in Half-Life implementierte Lösung wertet deshalb die clientseitige Sicht der Spielwelt zum Zeitpunkt des Feuerns auf Seiten des Servers aus. Man kann sich Lag Compensation als ein serverseitiges temporäres Zurückdrehen der Zeit vorstellen, um den Zustand wiederherzustellen, der für den Client zum fraglichen Zeitpunkt existiert haben muss.

Serverseitiger Lag-Compensation-Algorithmus vor Ausführen eines User-Kommandos:

1. die Latenz des Spielers möglichst genau bestimmen
2. die Liste der Spielwelt-Updates durchsuchen, die an den Spieler geschickt wurden, um das letzte Update zu finden, das an den Spieler geschickt und empfangen wurde, bevor sein Kommando ausgeführt wurde
3. die anderen Spieler anhand dieses Updates an die Position zurückbewegen, an der sie zum Zeitpunkt des Kommandos waren. Dabei die Latenz und das verwendete Darstellungsmodell des Clients berücksichtigen.
4. das Kommando des Users ausführen
5. alle Entitäten wieder an ihre tatsächliche / aktuelle Position zurückbewegen

In Schritt 3 ist zu beachten, dass dieses Zurückbewegen auch eine Rückführung anderer Variablen beinhalten kann, wie z.B. ob ein Spieler geduckt war.

Bewertung:
Lag Compensation ermöglicht es, dass Spieler in Half-Life direkt auf ihre Gegner zielen können, ohne ihre momentane Latenz berücksichtigen zu müssen. Das Spiel wird dadurch fairer und realistischer, bringt aber trotzdem einige Paradoxien mit sich, die im folgenden Abschnitt erläutert werden. Studien [Henderson (2003)] ergaben, dass Spieler in Half-Life ihre eigene Latenz ab ca. 250 ms überhaupt erst wahrnehmen können. Der Erfolg eines Spielers sinkt bei steigender Latenz nur sehr leicht. Der Nachteil ergibt sich durch das leicht erschwerte Zielen mit Projektilwaffen. Diese unterliegen bei Half-Life (wie auch bei allen anderen aktuellen Shootern) nicht der Client-Side-Prediction. Zwar wird das Mündungsfeuer lokal sofort angezeigt, das eigentliche Projektil wird aber serverseitig verwaltet, und fliegt deshalb erst los, nachdem das entsprechende Kommando beim Server eingetroffen und die dazugehörige Bestätigung beim Client angekommen ist.

Paradoxien durch Lag Compensation

Bei Lag Compensation sind die latenzbedingten Inkosistenzen eher für den Spieler ersichtlich, der beschossen wird. Als Beispiel dient die folgende Situation:
Ein Spieler H mit hohem Ping schießt auf einen Spieler L mit niedrigem Ping. Spieler H hat eine direkte Sicht auf sein Ziel und drückt den Feuerknopf. Bis das Kommando beim Server angekommen ist, ist Spieler L schon hinter einer Mauer verschwunden oder duckt sich hinter einer Kiste. Spieler L wird erst getroffen, sobald das Feuer-Kommando von Spieler H beim Server eintrifft. Eine zusätzliche zeitliche Verzögerung ergibt sich durch die Netzwerklatenz zwischen Server und Spieler L. Für Spieler L sieht es durch die Verzögerung nun so aus, als ob er um-die-Ecke bzw. durch die Wand erschossen worden sei.
Dieses Szenario ist bei einer ausreichend hohen Latenz von etwa 500 ms durchaus denkbar, stellt aber den Worst-Case dar. In den meisten Situationen wird dieser Effekt abgeschwächt. Wenn zwei Spieler frontal aufeinander zu laufen und dabei feuern, wird die Lag Compensation bei der Auswertung den getroffenen Spieler auf einer Linie mit der Blickrichtung zurücksetzen, wodurch keine offensichtlichen Inkonsistenzen entstehen.
Ein weiteres Paradoxon ergibt sich dadurch, dass der Blickwinkel des Schützen auf einen Punkt zielt, an dem sich der getroffene Spieler meist nicht mehr befindet. Der Schütze scheint für sein Opfer also leicht vorbeizuzielen. Dieser Effekt ist am stärksten, wenn sich das Opfer mit maximaler Geschwindigkeit rechtwinklig zur Blickrichtung des Schützen bewegt.

2.8.3 First Person Shooter: Unreal Tournament

Als dritter und letzter großer Vertreter der Multiplayer Egoshooter sei an dieser Stelle die *Unreal Tournament* Serie aus dem Hause *Digital Extremes / Epic Games* erwähnt. Der erste Teil erschien 1999, und wird heute oft *UT99* genannt, um Verwechslungen mit seinen Nachfolgern *UT 2003* und *UT 2004* zu vermeiden.
UT99 nutzte Client Side Prediction in Verbindung mit Extrapolation zum verzögerungsfreien Steuern der Avatare, verzichtete aber auf Lag Compensation. Unreal Tournament setzt stark auf eine möglichst flexibel und leicht zu modifizierende Spiellogik, um begabten Fans das Erstellen von kostenlosen Modifikationen („Mods") zu ermöglichen. Während die Grafikengine aus Gründen maximaler Performance hardwarenah in C und Assembler geschrieben ist, wird die Spiellogik in einer eigens entwickelten objektorientierten Skriptsprache namens *UnrealScript* programmiert. Auch die ursprüngliche Trefferauswertung wurde komplett in UnrealScript implementiert. Ein Jahr nach der Veröffentlichung von UT99 entwickelte der Community-Entwickler John Fraser eine Modifikation namens *Zeroping* [Fraser (2000)], die clientseitige Trefferauswertung in UT99 realisierte. In der letzten Version wurde dabei auch das folgende Problem behandelt, das durch Paketverlust entstehen kann: Falls aufgrund von Verbindungsproblemen eine Zeit lang keine Updates vom Server empfangen wurden, friert das Bild ein, wodurch der Spieler leichtes Spiel beim Zielen hat. Um Clients nicht von Paketverlust profitieren zu lassen, wurde deshalb ein zeitlicher Schwellwert eingebaut. Übersteigt der Zeitstempel einer clientseitigen Treffernachricht einen bestimmten Wert, wird die Nachricht verworfen, und es wird vorübergehend auf die ursprüngliche serverseitige Trefferauswertung umgeschaltet.
UT2003 und UT2004 wurden offiziell mit clientseitiger Trefferauswertung (Lag Compensation) ausgeliefert [Beigbeder u. a. (2004a)]. Wie in der Zeroping-Modifikation werden dabei nur Instant-Hit-Waffen berücksichtigt. Projektilwaffen wie Raketenwerfer und Pulsegun schießen weiterhin mit latenzbedingter Verzögerung, da ihre Projektile serverseitige Objekte sind, die erst nach Eintreffen des Feuer-Kommandos generiert werden. UT überträgt dabei nur einmalig die Position, den Geschwindigkeitsvektor und Zeitstempel des abgefeuerten Projektils. Die komplette Flugbahn wird nach dem Empfang clientseitig extrapoliert, wobei auch eine clientseitige Kollisionsabfrage stattfindet. Die clientseitig dargestellte Flugbahn dient jedoch nur der Visualisierung, und hat keinen Einfluss auf die Spiellogik. Es kann dabei aufgrund der latenzbedingten Inkonsistenzen vorkommen, dass ein Projektil aus Sicht eines Clients einen Gegner getroffen hat, aber aus Serversicht verfehlt hat, und noch bis zur Mapgrenze weiterfliegt. Auch bei UT gilt das Prinzip des autoritativen Servers, dessen Sicht als Referenz für alle Spieler gilt. Die Schadenspunkte sowie die physikalischen Effekte eines Projektileinschlags werden serverseitig berechnet und an die Clients weitergeleitet [Lambert (2001)]. Zur Reduzierung des Datenverkehrs überträgt der Server nur Daten zu Objekten, die innerhalb der Sichtweite des entsprechenden Clients liegen.

2.8.4 Sportspiel: Online Madden NFL Football

Online Madden NFL Football ist mit 40 Millionen verkauften Einheiten die zur Zeit weltweit erfolgreichste Serie im Segment der Football-Sportspiele. Wie bei fast allen Sportspielen wird auch hier das direkte Steuerungsmodell (siehe 3.2.2) genutzt. Der Sourcecode von *Online Madden NFL Football* wurde bisher nicht freigegeben, aber der Netzwerkcode sowie die Auswirkungen von Latenz wurden experimentell analysiert [Nichols und Claypool (2004)]. Madden NFL nutzt Local Lag in Verbindung mit Timewarp, um einen Kompromiss zwischen Spielbarkeit und Konsistenz zu erreichen. Dabei sind relativ hohe Latenzen von bis zu 500 ms RTT möglich, ohne von Spielern überhaupt wahrgenommen zu werden. Dem Spiel kommt dabei der relativ hohe Realismusgrad zugute, der aufgrund der Masseträgheit allzu abrupte Richtungswechsel verbietet. Jenseits der Grenze von 500 ms wirkt sich die Latenz immer deutlicher auf den Spielspaß und die Spielleistung aus. Mit steigender Verzögerung wird es im Hinblick auf das Passspiel immer schwieriger für den Spieler, günstige Momente abzupassen, da Lücken in der Abwehr meist nur für kurze Momente existieren. Auch das Steuern der Bewegungsrichtung wird träge, so dass Spieler immer häufiger gegen andere Spieler oder ins Seitenaus laufen.

2.8.5 Echtzeit-Strategie: Age of Empires III

Age of Empires des Softwarehauses *Ensemble Studios* ist mit ca. 16 Millionen verkauften Einheiten eine der weltweit erfolgreichsten Echtzeitstrategie-Serien. Der Sourcecode ist geschlossen, aber die Entwickler veröffentlichten detaillierte Angaben zur verwendeten Multiplayer-Technologie [Bettner und Terrano (2001)].
In modernen Echtzeitstrategiespielen werden hunderte oder sogar tausende von Einheiten über die Spielwelt bewegt. Mit dem bisherigen Ansatz, zu jedem Objekt periodisch die aktuelle Position, Geschwindigkeit und Ausrichtung zu übertragen, wären auf einem durchschnittlichen Zielsystem maximal 250 Objekte möglich. Stattdessen werden im RTS-Genre nur die reinen Spieler-Kommandos übertragen, wie z.B. *bewege Einheit E an Position X,Y*. Der Netcode muss nun sicherstellen, dass die exakt gleiche Spiellogik synchron bei allen Spielern ausgeführt wird, um die Konsistenz zu garantieren.

Architektur

AoE basiert auf einer Peer-to-Peer-Architektur. Normalerweise wird in kommerziellen Spielen eine Client/Server Architektur genutzt, da so bessere Maßnahmen gegen Cheating getroffen werden können. Das Genre der Echtzeit-Strategiespiele bildet hier die Ausnahme, weil hier eine absolute Konsistenz durch vollständige Synchronisation realisiert werden muss, und Inkonsistenzen sofort erkannt werden. Es ist dazu notwendig, den kompletten Datenverkehr als *reliable* zu markieren, und die Übertragung jedes einzelnen Kommandos von der

Gegenseite bestätigen zu lassen. Ein böswilliger Client müsste auch die Berechnung des Spielverlaufs bei den entfernten Teilnehmern manipulieren, um sich Vorteile zu verschaffen. Das Cheating wird durch diesen Umstand quasi unmöglich gemacht. In AoE wird der Spielverlauf angehalten, bis ein Kommando von allen Kommunikationsteilnehmern bestätigt wird. Werden Inkonsistenzen entdeckt, bricht das Spiel komplett ab.

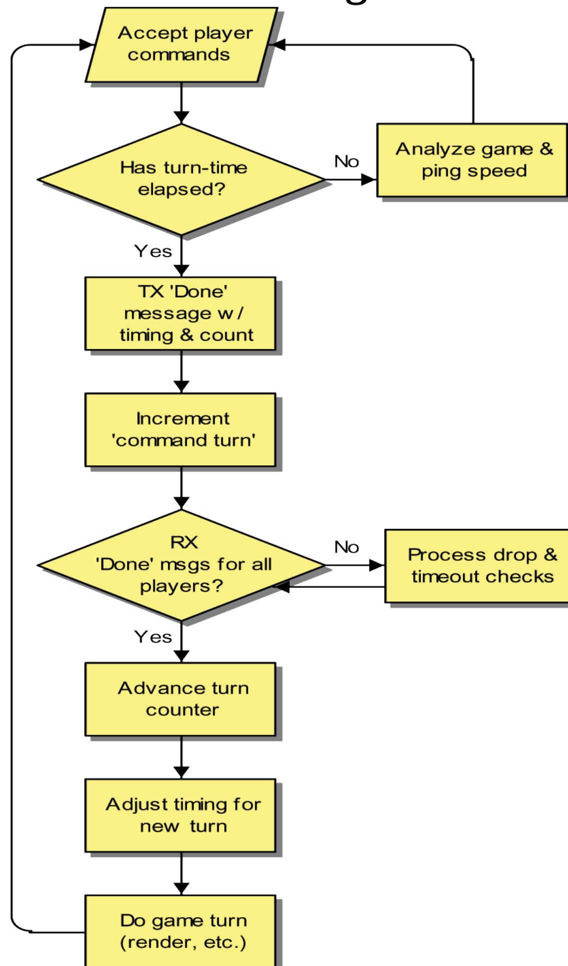


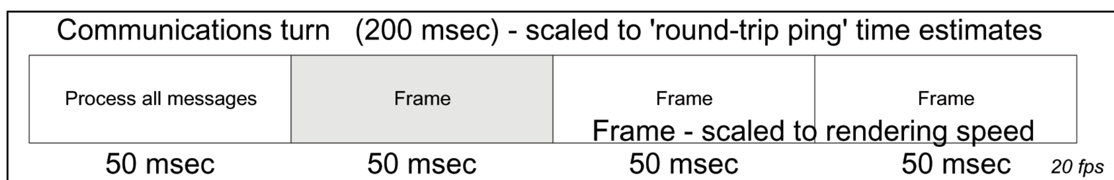
Abbildung 2.20: Abarbeitung von 'Turns' in Age of Empires [Bettner und Terrano (2001)]

Synchronisation

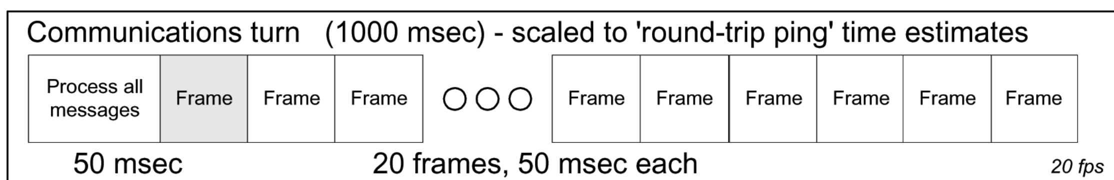
AoE benutzt zur Synchronisation einen Algorithmus, der mit der in Abschnitt 2.5.3 vorgestellten Bucket Synchronisation vergleichbar ist. Der Spielverlauf wird dabei in Zeitfenster namens *Turns* unterteilt. Ein Turn ist so lang wie die durchschnittliche RTT der langsams-

ten Verbindung unter allen beteiligten Peers. Alle Kommandos werden nicht sofort, sondern erst zwei Turns in der Zukunft ausgeführt. Dadurch bleibt im Normalfall genug Zeit, um das Kommando an alle anderen Peers zu senden und von allen eine Bestätigung zu empfangen. Falls aufgrund von Netzwerkproblemen nicht alle Bestätigungen rechtzeitig angekommen sind, hält das Spiel an, und es kommt zu einer kurzen Verzögerung bzw. Rucklern in der Darstellung. Hier besteht der Unterschied zur Bucket Synchronisation, bei der in diesen Fällen die Konsistenz zu Gunsten eines flüssigeren Spielerlebnisses geopfert wird. Die Länge eines Turns wird ständig analog zu den variierenden Netzwerklatenzen angepasst. Ein Turn wird in einzelne Frames gleicher Länge eingeteilt, über die die Bewegung der Objekte interpoliert wird. Die Framerate ist bei allen beteiligten Peers identisch und richtet sich nach dem langsamsten Teilnehmer (siehe Abbildung 2.21).

A Single Communication Turn



High Internet Latency with normal machine performance



Poor machine performance with normal latency

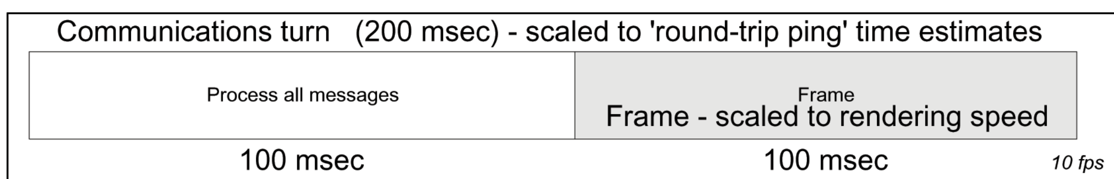


Abbildung 2.21: Aufbau von ’Turns’ in Age of Empires [Bettner und Terrano (2001)]

Latenzen

Das RTS-Genre ist vergleichsweise sehr genügsam, was die Anforderungen an die Netzwerklatenz betrifft. Die Ursache dafür liegt darin, dass hier die Reaktionsschnelligkeit nicht im Mittelpunkt steht, und die Einheiten nicht direkt gesteuert werden. Anaysen des RTS-Titels *Warcraft III* [Sheldon u. a. (2003)] haben gezeigt, dass auch Latenzen von mehreren Sekunden keinen signifikanten Einfluss auf die Leistung eines Spielers hat. Um eine Strategie auszuführen, brauchen Einheiten in RTS-Spielen mehrere Sekunden oder sogar Minuten. Eine Verzögerung von weniger als einer Sekunde wirkt sich dementsprehend gering auf den Spielverlauf aus.
In Usability-Tests [Bettner und Terrano (2001)] der Ensemble Studios nahmen die Spieler Latenzen erst ab etwa 250 ms wahr. Bei Werten von 250 - 500 ms wurde das Spiel noch als sehr gut spielbar empfunden. Jenseits der 500 ms war die Latenz zwar spürbar, aber die Spieler stellten sich darauf ein und entwickelten schnell ein Gefühl dafür, wann eine Einheit nach dem Klicken reagieren wird. Als dementsprechend störend wirkte sich deshalb ein ständiges Variieren der Verzögerung aus. Eine langsame, aber konstante Verzögerung von 500 ms wurde als deutlich angenehmer bewertet als eine Verzögerung, die ständig zwischen 80 und 500 ms schwankte.

2.9 Zusammenfassung der Analyse

Wir haben im Analyseteil verschiedene Multiplayer-Spielegenres und Mobilfunknetze vorgestellt. Weiter haben wir erkannt, dass die Netzwerklatenz das zentrale Problem actionlastiger Multiplayerspiele darstellt, und dass eine clientseitige Vorhersage der Spielsituation unverzichtbar ist, um trotz Latenz eine gute Spielbarkeit zu erhalten. Die dadurch entstehenden Inkonsistenzen können durch verschiedene Techniken reduziert oder kaschiert werden. Hierbei ist vor allem das genutzte Modell zur Darstellung verteilter Spielobjekte entscheidend. Hier stehen als grundlegende Techniken Interpolation und Extrapolation zur Verfügung. Im Bereich der Extrapolation ergeben sich verschiedene Variationen durch die Kombination von Extrapolations- und Konvergenzalgorithmen, sowie der Nutzung von Time Compensation. Jedes Darstellungsmodell stellt einen individuellen Kompromiss zwischen Konsistenz, flüssiger Bewegungsdarstellung, Gesamtverzögerung, Rechenbedarf und Datentransfervolumen dar. Welches Modell sich für welches Spiel am besten eignet, hängt stark von den Anforderungen des jeweiligen Spielgenres, dem Beschleunigungsverhalten des gesteuerten Spielobjekts, sowie der Performanz der Netzwerkverbindung ab. In jedem Darstellungsmodell können zur Optimierung im konkreten Anwendungsfall mehrere Parameter angepasst werden. Ein Ziel des praktischen Teils dieser Arbeit muss es deshalb sein, einem beliebigen Spielentwurf ein möglichst geeignetes Darstellungsmodell zuordnen zu können. In Abb. 2.22 werden die im Analyse-Kapitel beschriebenen Probleme und Techniken zur besseren Übersicht grafisch dargestellt.

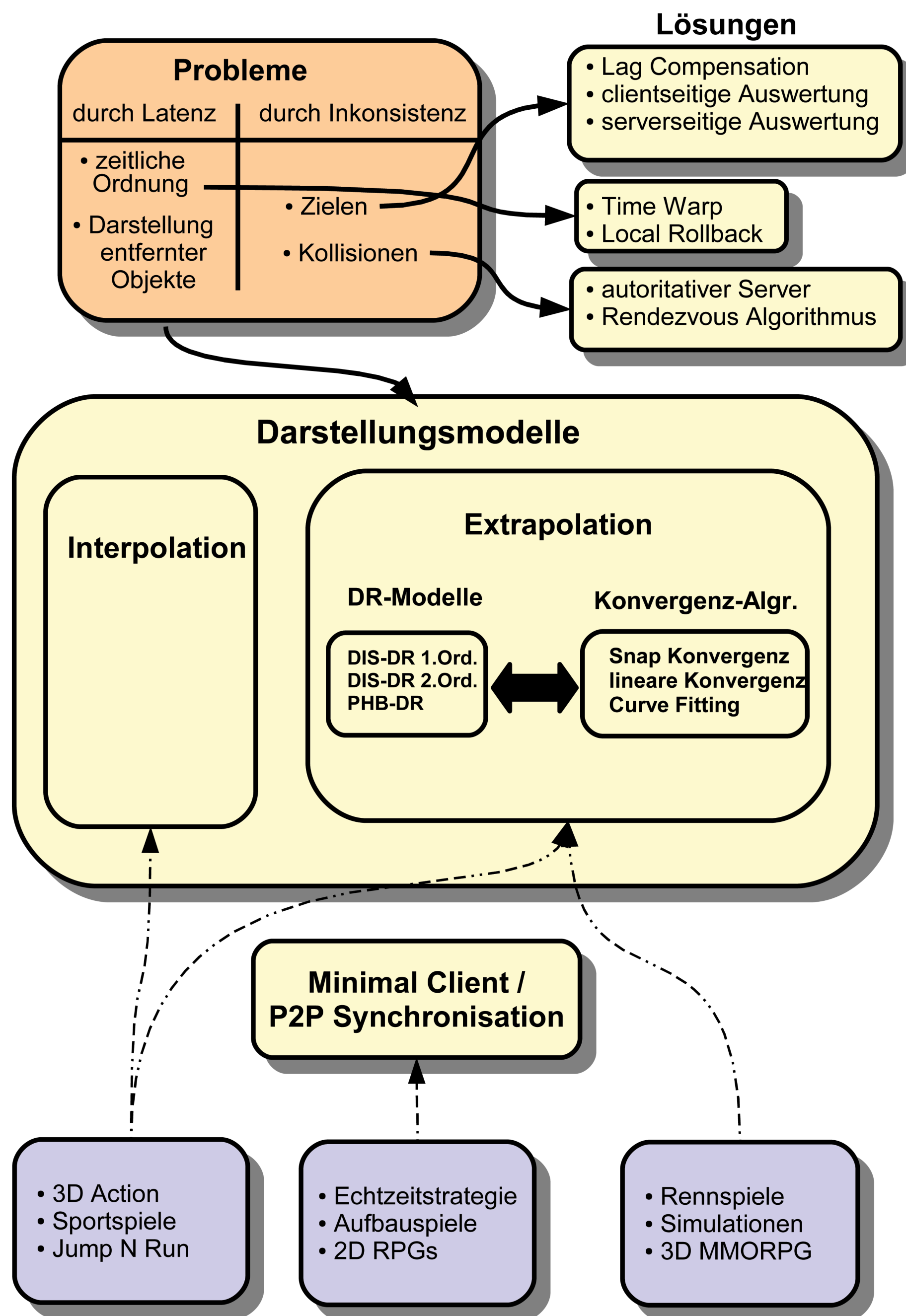


Abbildung 2.22: Übersicht: Probleme und Lösungsansätze in verteilten Echtzeitspielen

Kapitel 3

Entwurf und Realisierung

Im analytischen Teil dieser Arbeit haben wir Techniken zum Latenzausgleich vorgestellt, die sich in der Welt der Netzwerkspiele und der militärischen Simulationen bewährt haben. Da es dabei kein Verfahren gibt, das für alle Anwendungsfälle gleichermaßen geeignet ist, bietet sich deshalb an, diese Techniken zu Evaluationszwecken in einer maximal trivialen Spielumgebung zu implementieren. Dabei muss das Steuerungsverhalten des vom Spieler kontrollierten Spielobjekts konfigurierbar sein, um dem jeweils zu untersuchenden Spieletyp möglichst nahe zu kommen. Für die verschiedenen Genres soll jeweils dasjenige Darstellungsmodell gefunden werden, das unter den jeweiligen Netzwerkeigenschaften die geeignetsten Resultate in Bezug auf die Spielbarkeit liefert. Dazu ist es nötig, die Spielbarkeit anhand analytischer Daten zu erfassen und auszuwerten. Diese Daten umfassen beispielsweise das Transfervolumen der ausgetauschten Nutzdaten, oder den Positionsversatz in der lokalen Darstellung einer entfernten Entität.
Aufgrund der stark unterschiedlichen Quality-of-Service-Eigenschaften in mobilen Netzen kann beim Testen der Darstellungsmodelle nicht von konstanten Werten für Latenz, Paketverlust und Jitter ausgegangen werden. Diese Rahmenbedingungen variieren nicht nur im Hinblick auf die verschiedenen Mobilfunktechniken, sondern hängen auch von der momentanen Auslastung des Netzes, der Stärke des Funksignals und der Performanz des mobilen Endgeräts ab. Die Netzwerkeigenschaften sollten in unserer Testumgebung deshalb ebenfalls simuliert werden, und idealerweise frei konfigurierbar sein, um erstens von verlässlichen Werten ausgehen zu können, und um zweitens die Rahmenbedingungen verschiedenster Funkverbindungen nachbilden zu können. Übergeordnetes Ziel und Grundlage unserer Arbeit ist die Weiterentwicklung der Multiplayer-Plattform Exit Games Neutron, die wir im nun folgenden Unterkapitel vorstellen.

3.1 Neutron-Echtzeit-Plattform

Neutron wurde entwickelt, um Handyspiele und ähnliche mobile Applikationen möglichst schnell und einfach netzwerkfähig machen zu können. Entwickler binden dazu in ihr Projekt die Neutron-Funktionsbibliothek ein, die als J2ME und BREW-Version erhältlich ist. Über die Bibliothek wird ein Kommunikationskanal vom Handy zu einem von mehreren Neutron-Servern aufgebaut, die zur Zeit in Deutschland und in den USA betrieben werden. Neben der Bibliothek und den Servern umfasst die Neutron-Plattform als dritte Komponente das Webinterface *Neutron Control Center*, über das registrierte Entwickler die Neutron-bezogenen Parameter ihres Projekts individuell anpassen, und den Kommunikationsverlauf überwachen können.
Ein wesentliches Ziel bei der Entwicklung von Neutron war die maximale Kompatibilität: Spieler sollten mit unterschiedlichsten Handys und über alle Mobilfunkanbieter hinweg gegeneinander spielen können. Da die Unterstützung von Server-Sockets nicht im MIDP1.0-Standard enthalten war, wurde zum Datenaustausch auf das HTTP Protokoll als kleinsten gemeinsamen Nenner zurückgegriffen. Die Nutzung von HTTP implizierte, dass Nachrichten nicht unmittelbar empfangen werden konnten, sondern in regelmäßigen Abständen aktiv per Polling abgesammelt werden mussten. Die Polling-Frequenz kann dabei nicht beliebig hoch gewählt werden, da jeder Zugriff Datenverkehr und Prozessorlast verursacht. In der Praxis sind deshalb Polling-Intervalle von etwa 2 bis 4 Sekunden praktikabel. Alle zu kommunizierenden Spieleraktionen werden als *Events* bis zum nächsten Polling in einer Queue vorgehalten. Bis zum Datenaustausch vergehen also im Durchschnitt *Pollingintervall / 2* Sekunden bis zum Versenden, und entsprechend nochmals *Pollingintervall / 2* Sekunden bis zum Absammeln der Events durch den Mitspieler. Dazu addieren sich die Netzlatenzen vom Sender zum Neutron-Server, und vom Neutron-Server zum Gegenspieler. Je nach Verbindung und Konfiguration muss man in der Praxis bei der HTTP-Kommunikation mit etwa 3-6 Sekunden Gesamtverzögerung rechnen. Die Realisierung von Echtzeit-Spielen ist über HTTP damit praktisch unmöglich.
Mit Einführung des MIDP2.0 Standards und der zunehmenden Verbreitung der entsprechenden Handys ergeben sich heute für Neutron neue Erweiterungsmöglichkeiten, da u. a. die socketbasierte Datagramm-Kommunikation Teil des MIDP-Standards wurde. Die Neutron-Funktionsbibliothek kann nun um eine Datagramm-basierte Komponente erweitert werden, mit der auch Echtzeitspiele möglich werden.
Exit Games erweitert zum Zeitpunkt der Erstellung dieser Arbeit ihr Produktportfolio um eine Echtzeit-Plattform, bestehend aus der *Neutron Realtime Library* als neue Funktionsbibliothek und einer *Realtime Server* Komponente. Aufgrund der in 2.6.2.2 angesprochenen Nachteile eines Peer-to-Peer-Ansatzes wurde auch für die Neutron-Echtzeit-Plattform eine Client/Server-Architektur gewählt. Abbildung 3.1 zeigt die Gesamtarchitektur der Neutron-Echtzeit-Plattform zum Zeitpunkt des Beginns dieser Arbeit.

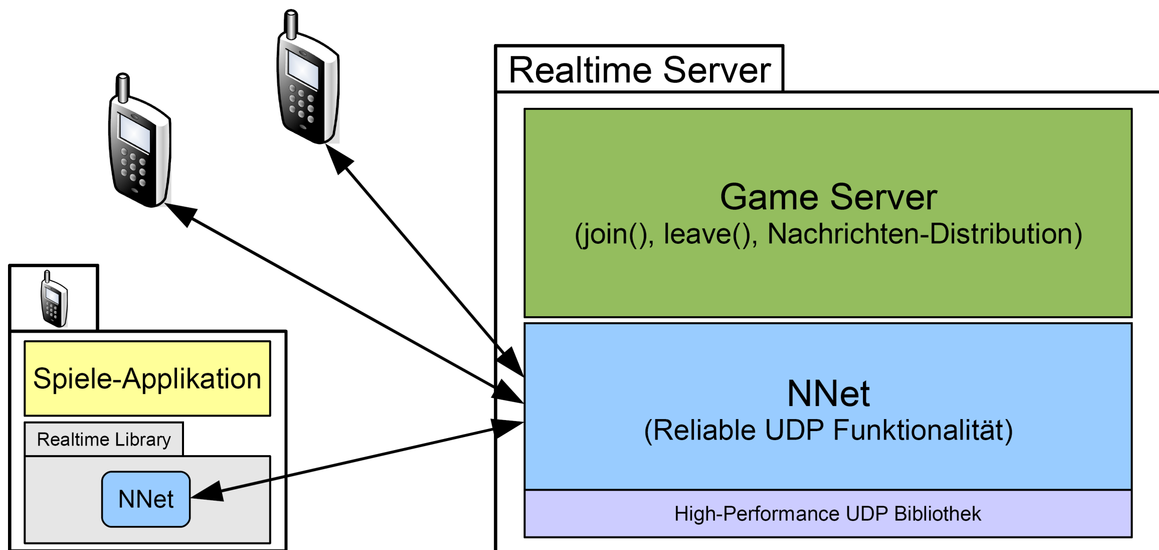


Abbildung 3.1: Gesamtarchitektur der Neutron-Echtzeit-Plattform

Neutron Realtime Library

Die Neutron Realtime Library setzt sich aus den Komponenten *NNet* und *NeutronListener* zusammen (siehe Abb. 3.2). Das NeutronListener-Interface ist dabei die Schnittstelle zwischen der Spiele-Applikation und der Neutron Realtime Library. Wichtigster Teil dieses Interfaces ist die Methode *eventAction*, die von der Neutron-Komponente aufgerufen wird, sobald neue Spieldaten empfangen wurden. Aufgrund des frühen Entwicklungsstadiums der Neutron-Echtzeit-Plattform ist diese Methode zur Zeit noch die einzige, die im NeutronListener-Interface vorgeschrieben wird. Es ist geplant, die übrigen Funktionen der HTTP-Version nach und nach zu integrieren.

NNet

NNet ist eine von Exit Games implementierte Reliable-UDP-Schicht, die für eine hochperformante Echtzeitkommunikation konzipiert wurde und auf dem in Sektion 2.6.1 vorgestellten ENet basiert. NNet-Portierungen sind in J2ME und J2SE verfügbar, wobei weitere Versionen für BREW und Win32 folgen sollen. Die Implementationen in J2ME und J2SE befanden sich während der Entstehung dieser Arbeit weiterhin in Entwicklung. Da die Relevanz für mobile Echtzeitspiele noch nicht einzuschätzen war, fehlen die in ENet enthaltenen Funktionen der Kanäle, Fragmentierung und Überlaststeuerung. Das Hauptziel von NNet war im ersten Schritt, eine möglichst simple und leichtgewichtige Netzwerkschicht zu realisieren, die UDP-Pakete auf Wunsch auch zuverlässig und unter Erhaltung der Sendereihenfolge versenden

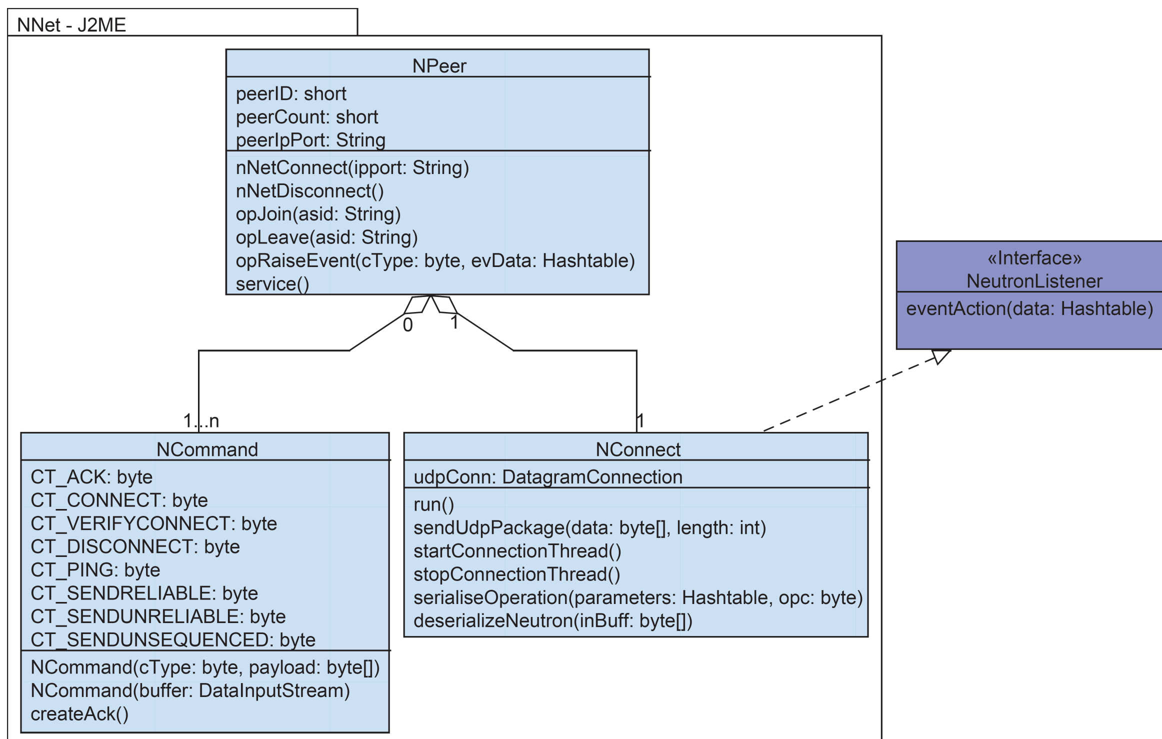


Abbildung 3.2: Neutron Realtime Library

kann. Die J2ME-Implementation von NNet besteht aus den drei Klassen NPeer, NConnect und NCommand (siehe Abb. 3.2).

- NPeer: NPeer bietet die grundlegenden Funktionen connect(), disconnect(), join(), leave(), sowie das Versenden von Nutzdaten mittels raiseEvent() an. Die Klasse kapselt die Funktionalität von NNet für den Anwendungsentwickler. Nutzdaten können als einfache unzuverlässige Datagramme, oder bei Bedarf auch zuverlässig und unter Erhaltung ihrer Reihenfolge versendet werden. Der Anwender muss im Gameloop der Spiele-Applikation die Methode *service()* der Klasse NPeer aufrufen, damit die Eingangsqueue synchron dazu abgearbeitet wird. Des weiteren reguliert service() das Zusammenfassen von mehreren ausgehenden Nachrichten zu größeren Paketen (siehe „Aggregation“ in Abschnitt 2.6.1).

- NCommands: Die Klasse NCommands repräsentiert die verschiedenen Nachrichtentypen, die in NNet ausgetauscht werden. Dies sind Protokollnachrichten wie ACK, CONNECT, VERIFYCONNECT, DISCONNECT und PING, sowie die für Nutzdaten wählbaren Versandarten SENDRELIABLE, SENDUNRELIABLE und SENDUNSEQUENCED.

- NConnect: NConnect ist ein Thread, der eine Datagramm-Verbindung verwaltet, über die auf die UDP- Transportschicht zugegriffen wird. Hier setzt NNet also letztendlich auf das UDP-Protokoll auf. NConnect ist für das Serialisieren und das Versenden der NCommands, und entsprechend auch für das Deserialisieren und Weiterleiten empfangener NCommands an den NeutronListener zuständig.

Realtime Server

Der Realtime Server besteht aus den Komponenten *Game Server* sowie einer speziellen Implementierung der NNet-Netzwerkschicht in der Sprache C++.
Die Standardversion des Neutron-Servers unterstützt einen Basissatz an Funktionalitäten, der schon für eine breite Anzahl an verschiedenen Spielen genügen sollte. Diese Server lassen sich allerdings nur beschränkt, in der Form von virtuellen Spielern (AI Bots) und virtuellen Spielleitern (Virtual Masters), erweitern. Um jedoch komplexere Spiele zu ermöglichen, werden aufwendigere, individuell konfigurierbare Funktionalitäten auf Seiten des Servers benötigt. Daher bietet Exit Games in seinem Produktportfolio auch spezielle Game Server an. Game Server implementieren weiterhin den Basissatz an Neutronfunktionen, und ermöglichen zusätzlich durch eine Server-SDK die individuelle Konfigurierbarkeit auf die Anforderungen des Spiels. Um auch hier den Spielentwicklern Arbeit abzunehmen, gibt es mehrere fertige Game Server Frameworks für typische Spiele wie Trading Card oder Trivia Games. Der in der Neutron-Echtzeit-Plattform verwendete Game Server stellt jedoch einen Sonderfall dar. Es handelt sich dabei um einen Prototypen, der momentan nur die Basisfunktionalität Neutrons anbietet, bestehend aus dem Beitreten und Verlassen eines Spiels, sowie der Distribution von Nachrichten. Oberstes Ziel ist im ersten Ansatz die Optimierung der Geschwindigkeit der Nachrichtendistribution. Der Realtime Game Server basiert deshalb ebenso wie die Client-Bibliothek auf NNet als Reliable-UDP-Schicht. Für einen besonders optimierten Zugriff von NNet auf die Transportschicht wird eine spezielle hochperformante UDP Bibliothek eines Drittanbieters genutzt, die hardwarenah in C implementiert wurde. Die Verbindung des Game Servers mit NNet erfolgt durch zusätzlichen Code in C#, der unter Verwendung des .NET-Frameworks eingebunden wurde.

Implementierung der Zeitsynchronisation

Wir haben in Kapitel 2.5.4 die Notwendigkeit einer gemeinsamen Uhrzeit erläutert und Techniken zur Zeitsynchronisation vorgestellt. Jeder Client muss von ihm ausgelöste zeitkritische Events mit einem Zeitstempel versehen, der einen Zeitpunkt auf einer gemeinsamen Zeitachse beschreibt. Diese gemeinsame Spielzeit (beispielsweise in Form der vergangenen Millisekunden seit Spielstart) wird üblicherweise vom Server vorgegeben, und ist demensprechend direkt von der Systemzeit des Servers abhängig. Der Client muss deshalb

jederzeit die Abweichung seiner eigenen Systemzeit relativ zur Serverzeit kennen. Die Verwendung des Push-Algorithmus' setzt eine konstante Latenzzeit voraus, die in unserem Kontext nur über die indirekte Messung der RTT ermittelt werden kann, was wiederum dem Pull-Algorithmus entspricht. Um den Synchronisationsfehler möglichst gering zu halten, sollte der Pull-Algorithmus in Abständen von etwa 10 Sekunden angestoßen werden.
Beim Bearbeiten der Zeitanfragen muss sichergestellt sein, dass der Server seine eigene Bearbeitungszeit I misst, um die Serverzeit TS_R beim Eintreffen der Antwort möglichst unverfälscht errechnen zu können. Dazu muss die Zeitdifferenz vom Erhalt der Anfrage bis zum Absenden der Antwort gemessen und der Antwort beigefügt werden. Je tiefer diese Zeitmessung im Schichtenmodell der Echtzeit-Bibliothek erfolgt, desto geringer wird der zu erwartende Fehler sein, da Events von NNet aus Performancegründen in Queues vorgehalten und zusammengefasst werden.
Dieser Ansatz einer Zeitsynchronisation wurde Exit Games während der Erstellung dieser Arbeit vorgeschlagen, und wurde noch vor Abschluss in die Neutron-Echtzeit-Plattform integriert. Die Neutron Realtime Library berechnet nun permanent die Abweichung der lokalen Systemzeit zur Serverzeit, und hält den aktuellsten Wert in der Instanzvariable *serverTimeOffset* der Klasse NPeer vor. Der Zugriff auf diese Variable ermöglicht dem Spieleentwickler die Vergabe von Zeitstempeln für Ereignisse in absoluter Spielzeit.

3.2 Realtime Simulator

Bei der Entwicklung unserer Testapplikation mussten zunächst mehrere Fragen geklärt werden: Es lag nahe, dass wir die wichtigsten Technologien des analytischen Teils implementieren und unter möglichst realistischen Bedingungen testen müssen. Optimalerweise sollte dabei auch die aktuellste Version der Neutron Realtime Library benutzt werden, um deren Leistungsfähigkeit zu testen und eventuelle Unzulänglichkeiten erkennen zu können. Im Interesse von Exit Games lag diesbezüglich auch ein Showcase, der potentiellen Kunden zu Demonstrationzwecken auf mobilen Endgeräten präsentiert werden könnte. Andererseits sollte in unserer Testapplikation auch ein Einsatz unter optimalen Bedingungen möglich sein, um jede Technik frei von zusätzlichen Fehler- und Verzögerungsquellen untersuchen zu können. In einer latenzfreien Umgebung wäre es möglich, die zu erwartenden Rahmenbedingungen (z. B. Latenz und Paketverlust) der verschiedenen Mobilfunknetze zu simulieren. Dadurch würden nicht nur Zeit und Verbindungskosten gespart, sondern auch exaktere Ergebnisse erzielt, da die netzbedingten Latenzen nicht unserer Kontrolle unterliegen, und insbesondere asymmetrische Latenzen nicht ohne weiteres erkannt werden können. Eine latenzfreie Umgebung würde uns außerdem einen exakteren Vergleich der Darstellungsmodelle ermöglichen: Wir könnten zu jedem Zeitpunkt (d. h. in jedem einzelnen Frame) exakt bestimmen, wie weit die Position einer entfernten Entität von ihrer tatsächlichen momentanen Position auf dem Bildschirm des Gegners abweicht.

Um beide Anforderungen erfüllen zu können, entschieden wir uns für ein Implementieren der Testapplikation unter J2ME und MIDP2.0. Unter Benutzung der Neutron Realtime Library ist dieser Programmcode ohne weitere Modifikationen auf allen aktuellen Handys lauffähig. Um zu Testzwecken die Latenz zu minimieren, können auf einem lokalen Rechner zwei Instanzen eines J2ME Emulators gestartet werden. Erste Tests ergaben, dass unter Verwendung der Neutron Realtime Library stark schwankende Latenzen von etwa 150 - 400 ms erzielt wurden. Diese Werte waren unerwartet hoch und konnten nur mit dem frühen Entwicklungsstadium der Neutron Realtime Library erklärt werden, das auch die Ursache für zahlreiche Serverabstürze war. Um verlässliche Testergebnisse garantieren und unterbrechungsfrei weiterentwickeln zu können, war es notwendig, uns von der Neutron Realtime Library unabhängig zu machen. Gleichzeitig sollte ein späterer Wechsel zurück zur Neutron Realtime Library mit möglichst geringem Aufwand möglich sein. Wir entschieden uns deshalb für die Implementierung einer zusätzlichen Abstraktionsschicht zur Kapselung der Netzwerkverbindung.

3.2.1 Netzwerk-Abstraktionsschicht

Die Schicht erlaubt es uns, jederzeit leicht zwischen einer Netzwerkkommunikation über die Neutron-Echtzeit-Plattform oder einer einfachen Peer-to-Peer-Verbindung wechseln zu können. Für unsere Testzwecke waren nur zwei grundlegende Funktionen notwendig:

1. Verbindungsauf- /abbau
2. Austausch von Events gemäß des Neutron-Serialisierungs-Protokolls

Wir definierten dazu eine Oberklasse *ConnectionManager*, die drei Methoden mit der entsprechenden Funktionalität beinhaltet, und eine Referenz auf die zentrale Spielklasse besitzt, um deren Callback-Methoden aufrufen zu können. Erbende Klassen von ConnectionManager sind zum einen der *NeutronManager*, der die Kommunikation an die Neutron Realtime Library weiterleitet. Als Alternative implementieren wir nun außerdem die Klasse *DatagramManager*, die eine einfache Peer-to-Peer-Verbindung zu einer beliebigen IP-Adresse aufbaut. Der DatagramManager hält sich intern an das Neutron-Serialisierungsprotokoll, sodass der Nachrichtenaustausch kompatibel zu Neutron bleibt. Die Nachrichten werden im DatagramManager im Gegensatz zum NeutronManager aber nicht in einer Warteschlange vorgehalten, sondern sofort abgeschickt. Damit entfernen wir uns während der Testläufe ein Stück weit von der Realität, ermöglichen uns aber eine unverfälschte Analyse der implementierten Techniken. Die so gewonnenen Erkenntnisse können so später möglicherweise helfen, die Neutron Realtime Library zu optimieren.

Vorteile:

- Unabhängigkeit von der Neutron-Echtzeit-Plattform
- bessere Kontrolle über Simulationsparameter durch Eliminieren der Netzwerk-Latenz
- kann helfen, Unzulänglichkeiten in der Neutron Realtime Library zu erkennen

Nachteile:

- NNet-Schicht fehlt, daher nur einfache UDP-Funktionalität
- kein Matchmaking, nur 2-Spieler-Spiele möglich
- verfälschtes Latenzverhalten durch fehlende Ein-/Ausgangs-Queue

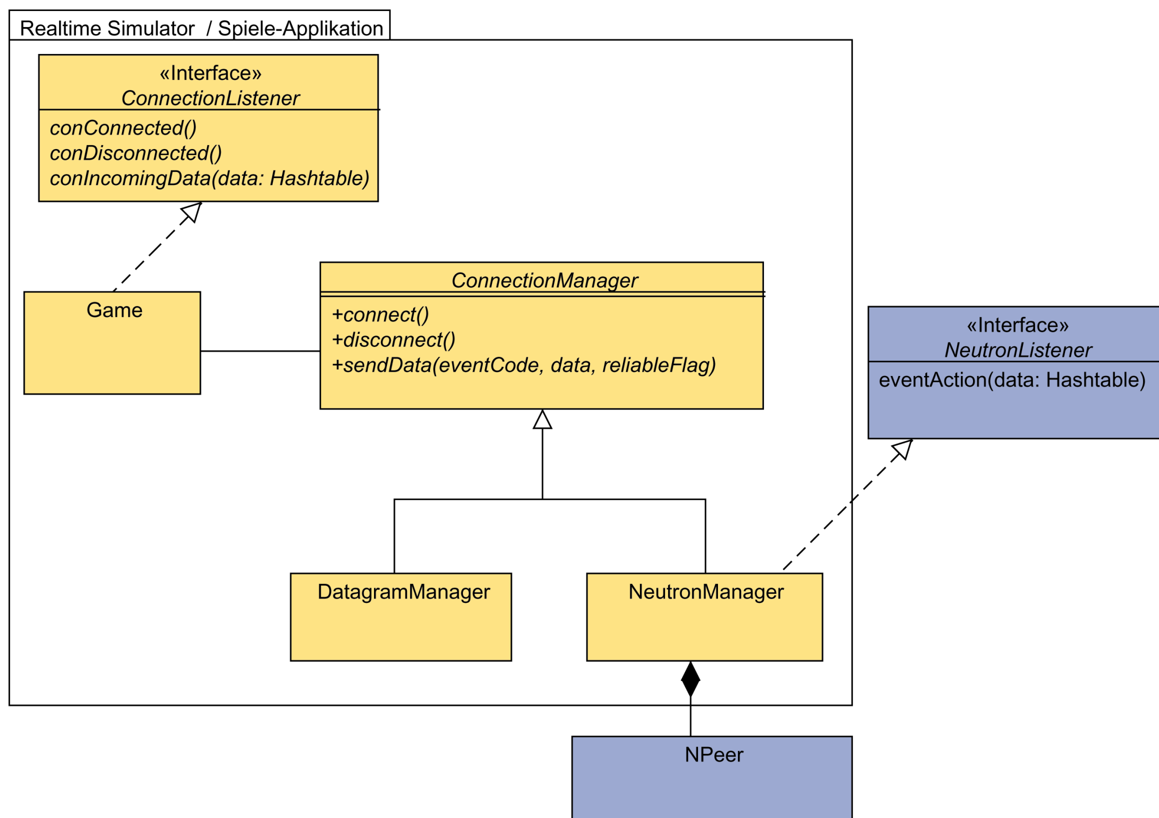


Abbildung 3.3: Netzwerk-Abstraktionsschicht durch ConnectionManager

Abbildung 3.3 zeigt ein Klassendiagramm der Klassen, die an der Netzwerk-Abstraktionsschicht beteiligt sind. Zentrale Klasse unserer Applikation ist die Game-Klasse,

in der der Gameloop angestoßen und die Grafik dargestellt wird. Das *ConnectionListener* Interface ist in unserem Modell nötig, um Callback-Methoden nach einem erfolgreichen Verbindungsaufbau und Datentransfer aufrufen zu können, damit in der Game-Klasse Rückmeldungen an den Benutzer ausgegeben und ggf. Zustandsvariablen gesetzt werden können.

3.2.2 Steuerungsmodelle

Um einen möglichst großen Teil verschiedener Spielegenres simulieren zu können, war es nötig, verschiedene typische Steuerungsmodelle in die Applikation zu integrieren. Die Interaktion in Echtzeitspielen lässt sich fast immer einem der folgenden Modellen zuordnen:

Direkte Steuerung

Die direkte Steuerung ist das Prinzip, das in den meisten Arcade-Automaten Verwendung findet: Das Spielobjekt wird aus der Vogelperspektive oder Seitenansicht gesteuert. Die Perspektive ändert sich dabei nie, deshalb kann die Bewegungsrichtung direkt der Steuerungsrichtung zugeordnet werden. Bei digitalen Eingabegeräten sind somit maximal 8 verschiedene Bewegungsrichtungen möglich.

Spielgenres:

- Sportspiele (Pro Evolution Soccer)
- 2D-Jump'n'Run (Super Mario Bros.)
- Beat'em Up (Mortal Kombat)
- Puzzlespiele (Tetris)
- Arcade-Actionspiele (Bomberman)

3D-Action

Dieses Modell ist in fast allen aktuellen actionorientierten PC- und Konsolenspielen zu finden. Der Spieler sieht die Spielwelt entweder in der Perspektive seiner Spielfigur („Ego-Perspektive“), oder folgt ihr in einer etwas erhöhten Perspektive (3rd Person View). Blickwinkel und Bewegungsrichtung der Spielfigur können durch stufenlose Drehung frei gewählt werden. Neben dem Vor- und Rückwärtslaufen sind normalerweise auch seitliche Schritte möglich, ohne die Blickrichtung ändern zu müssen („strafing“).

Spielgenres:

- 3D-Shooter (Counter-Strike)
- 3D-Action Adventures (Tomb Raider)
- 3D-Rollenspiele (World of Warcraft)
- 3D-Jump'n'Runs (Sonic Heroes)

Simulationen

In Simulationsspielen hängt das Steuerungsmodell stark vom simulierten Objekt ab, kann aber meist grob als eingeschränkte Variante des 3D-Action-Modells beschrieben werden. Die Bewegung ist meist nur in Blickrichtung und teilweise auch rückwärts möglich.

Spielgenres:

- Flugsimulationen (Falcon)
- Rennspiele (Need for Speed)
- Weltraum-Action (Asteroids, Wing Commander)
- U-Boot-Simulationen (Silent Hunter)

Point & Click

Dieses Steuerungsmodell findet man hauptsächlich in Echtzeitstrategie-Spielen (*Realtime Strategy, RTS*) vor. Die zu bewegende Einheit wird vom Spieler mit einem Cursor oder Mauszeiger angewählt, und danach wird die Position angeklickt, an die sich die Einheit bewegen soll. Die Einheit sucht sich mit Hilfe eines Wegfindungs-Algorithmus' automatisch den Weg zum Ziel. Es werden also eher Marschbefehle erteilt, anstatt das Objekt direkt zu steuern.

Spielgenres:

- Echtzeitstrategie (Warcraft III)
- Aufbauspiele (Die Siedler)
- Rollenspiele mit isometrischer Perspektive (Diablo, Neverwinter Nights)
- Adventures (The Secret of Monkey Island)

Implementierung der Steuerungsmodelle

Das Modell der Direkten Steuerung wurde von uns in der Klasse *SteeringModelDirect* abgebildet. In diesem Modell bewirkt das Drücken einer Richtungstaste eine sofortige Ausrichtung und Beschleunigung des Spielobjekts in die entsprechende Richtung. Das Modell *SteeringModelThrust* repräsentiert das Steuerungsmodell von Simulationsspielen und ist dem Modell *SteeringModel3dAction* sehr ähnlich, das zusätzlich um die Tasten zum seitlichen Ausweichen und den „Rückwärtsgang" erweitert wurde.
Auf eine Implementierung der Point & Click Steuerung haben wir verzichtet, weil sie einen Sonderfall darstellt: Bei Point & Click - Spielen kann in der Regel auf Extrapolationstechniken verzichtet werden, da es ausreicht, einfache Kommandos zu übertragen (siehe Abschnitt 2.8.5 zu Age of Empires III).

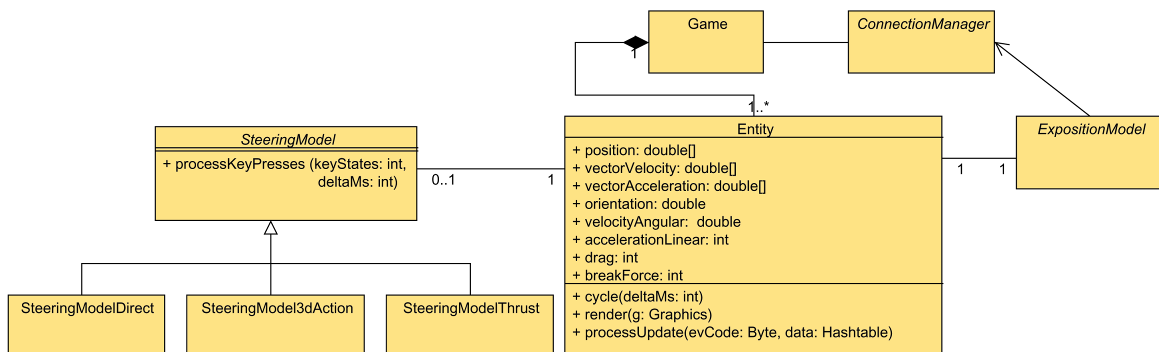


Abbildung 3.4: Unterstützung verschiedener Steuerungsmodelle

Entity

Die Klasse *Entity* repräsentiert ein verteiltes Spielobjekt, das entweder vom lokalen oder von einem entfernten Spieler gesteuert wird. In ihr werden grundsätzliche Eigenschaften wie die aktuelle Position, Orientierung, Geschwindigkeit und Beschleunigung verwaltet. Weitere Eigenschaften sind: maximale lineare Beschleunigung, automatisches Bremsverhalten („break force") und geschwindigkeitsabhängige Reibungskraft („drag"). Wird die Entity vom lokalen Spieler gesteuert, ist ihr ein Steuerungsmodell zugeordnet, in dem in jedem Gameloop-Durchlauf die Tastatureingaben interpretiert werden. Im Steuerungsmodell ist abgebildet, wie sich die Tastatureingaben auf den aktuellen Gesamtbeschleunigungsvektor und die Orientierung auswirken. In der *cycle()*-Methode der Entity wird in jedem Gameloop-Durchlauf anhand der Dauer des letzten Durchlaufs berechnet, über welche Dauer sich dieser Beschleunigungsvektor auf die Geschwindigkeit, und die Geschwindigkeit sich wiederum auf die aktuelle Position auswirkt. Am Ende der cycle()-Methode steht also die Darstellungspo-

sition und -orientierung der Entity im aktuellen Frame fest, und kann im nächsten Schritt gerendert werden.

3.2.3 Darstellungsmodelle

Entfernte Entitäten der anderen Mitspieler werden nicht über ein Steuerungsmodell, sondern über ihr Darstellungsmodell bewegt. Die Bewegung geschieht dabei in zwei Schritten. Zunächst wird in jedem Gameloop-Durchlauf das Darstellungsmodell der Entity angestoßen, das zunächst prüft, ob für die Entity neue Update-Pakete eingegangen sind. Das Darstellungsmodell verwaltet diese Updates in einer eigenen Datenstruktur, und ist für das Löschen veralteter Updates zuständig. Falls neue Daten eingegangen sind, ruft das Darstellungsmodell die Methode *calculateConvergencePath()* des ihm zugeordneten Konvergenzalgorithmus' auf. In dieser Methode wird der Pfad berechnet, auf dem die Entity weiterbewegt wird, um sich der neuen Position anzunähern.

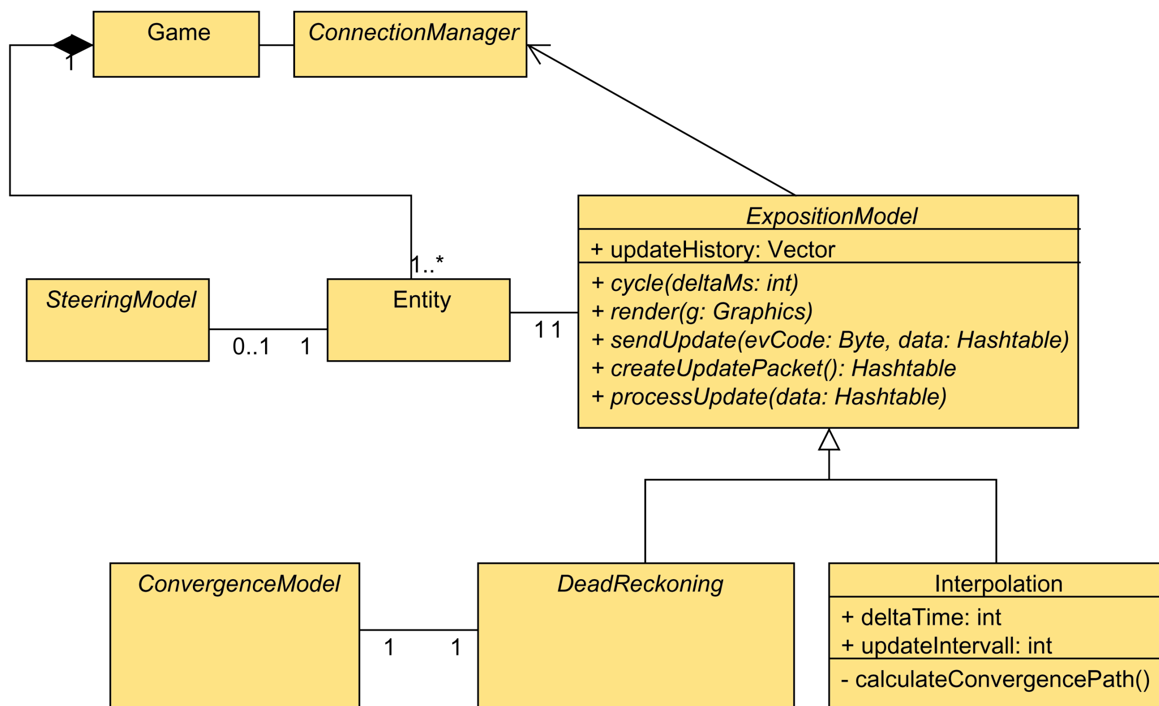


Abbildung 3.5: Klassendiagramm der Darstellungsmodelle

Konvergenzmodelle

Das Konvergenzmodell errechnet im Fall neu eingegangener Daten eine Zielposition („Konvergenzpunkt"), auf das die Entität zubewegt werden soll. Nachdem der Konvergenzpunkt

errechnet wurde, entscheidet der Konvergenzalgorithmus anhand der Dauer des letzten Gameloop-Durchlaufs, um wie viele Pixel die Entität weiterbewegt werden muss, um den Punkt in der gewünschten Konvergenzzeit zu erreichen. Im Falle von Dead Reckoning existieren zur Konvergenz eigene Klassen, die von der abstrakten Oberklasse *ConvergenceModel* erben. Die einfachste Variante stellt dabei die *Snap-Konvergenz* dar, die die Entität ruckartig auf die aktuelle Position verschiebt, sobald ein neues Update eingegangen ist.

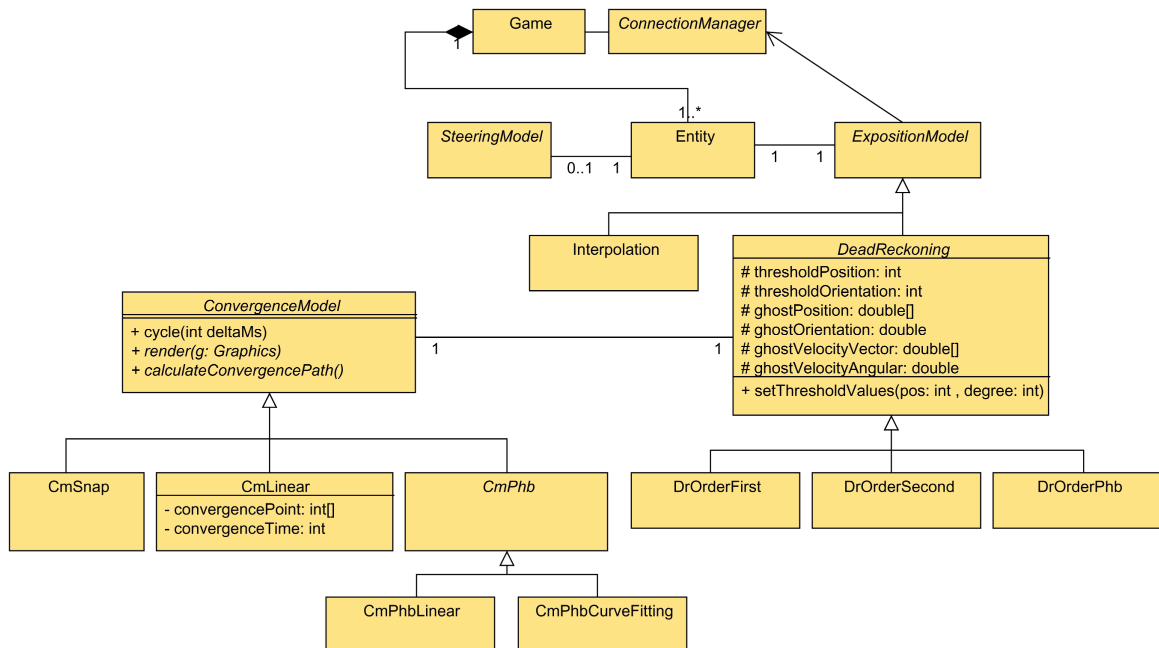


Abbildung 3.6: Kombinierbare Konvergenz- und DR-Modelle

Die lineare Konvergenz gemäß Sektion 2.7.1.3 wurde in der Klasse *CmLinear* realisiert. Im Falle von Interpolation ist nur die lineare Konvergenz sinnvoll, da der zu erreichende Punkt bereits feststeht, und deshalb keine Extrapolationsfehler auftreten können. Dieser Konvergenzalgorithmus ist deshalb fest in der Klasse *Interpolation* integriert und nicht austauschbar.

Ein Problem beim Implementieren der linearen Konvergenz ergab sich dadurch, dass sich der Konvergenzpunkt nach dem Zeitpunkt richtet, an dem „erwartungsgemäß“ das nächste Update eintrifft. Dieser Zeitpunkt ist bei Dead Reckoning jedoch nicht genau vorher zu bestimmen, da sich die Update-Frequenz nicht nach einem festen Takt, sondern nach dem Überschreiten des Schwellwerts richtet. Die Konvergenzzeit ist daher auch vom jeweils genutzten Dead-Reckoning-Darstellungsmodell abhängig, weshalb wir uns für eine freie Konfiguration der Konvergenzzeit im Optionsmenü entschieden haben. Der Konvergenzpunkt wird in unserer Testapplikation zunächst als roter Punkt dargestellt. Sobald die Entität den Konvergenzpunkt erreicht hat, wird die Geschwindigkeit und Orientierung der Entität gemäß

des DIS-Dead-Reckoning-Algorithmus auf die Werte des aktuellsten Updates gesetzt. Um diesen Zustandswechsel anzuzeigen, wechseln wir die Farbe des Konvergenzpunktes nach dessen Erreichen auf Grün.

Interpolation

Die Interpolation wurde mit dem in 2.7.3 beschriebenen Algorithmus implementiert, der eine Position History nutzt, um die Verfälschung der Bewegung möglichst gering zu halten. Diese Technik wird auch in den 3D-Shootern Half-Life und Counter-Strike eingesetzt. Im Optionsmenü sind zur Interpolation die Parameter *Deltazeit* und *Updateintervall* frei in Millisekunden wählbar. Beide Parameter müssen entsprechend der Netzwerklatenz so gewählt werden, dass immer mindestens ein neues Update eingetroffen ist, bis der Konvergenzalgorithmus die Entität zur letzten Zielposition bewegt hat. Liegt bei Erreichen der Zielposition kein neues Update vor, bleibt die Entität an der zuletzt empfangenen Position stehen, bis ein neues Update eintrifft. Um den Algorithmus zu visualisieren, werden in der Testapplikation alle in der Position History vorgehaltenen Updatepositionen als rote Punkte dargestellt. Das aktuellste Update wird als grüner Punkt gekennzeichnet (siehe Abbildung 3.7). Sobald die Entität die nächste Zielposition erreicht hat, ist das zugehörige Update veraltet, und wird aus der Position History gelöscht.

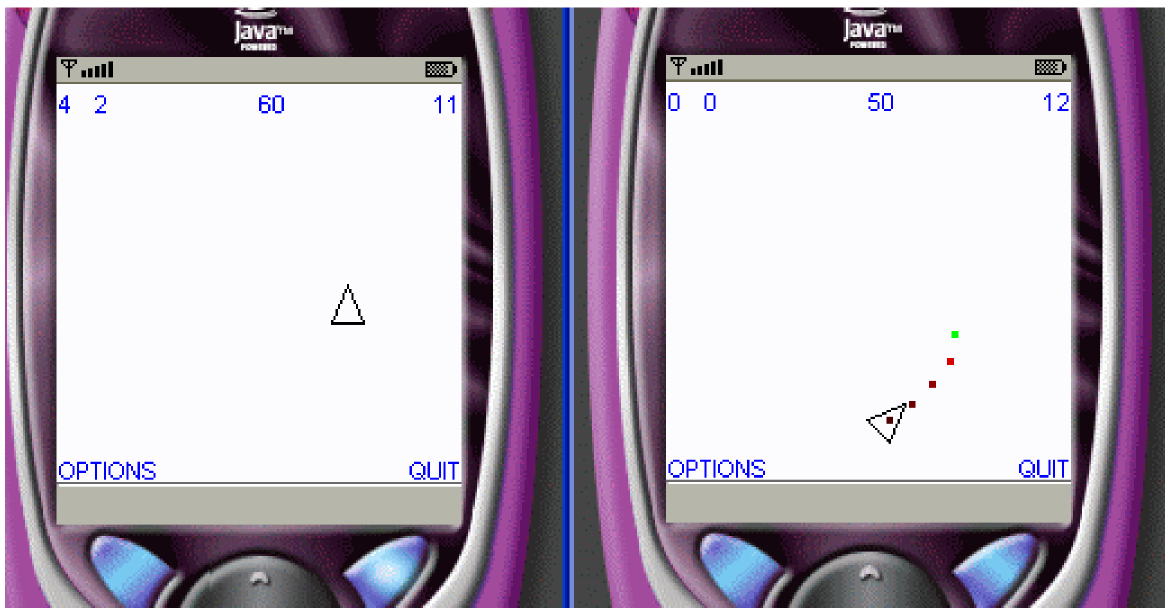


Abbildung 3.7: Interpolation mit Position History, links: lokaler Spieler

3.2.4 Betriebsmodi und Optionen

Startmenü

Nach dem Start des Realtime Simulators wird zunächst die gewünschte Netzwerkverbindung gewählt. Hier stehen die Modi *Neutron Realtime Library*, *Peer-to-Peer* und *Local Loopback* zur Auswahl (siehe Abb. 3.8). Im Peer-to-Peer-Modus ist die Eingabe einer IP-Adresse erforderlich, zu der die Verbindung aufgebaut werden soll, sowie unterschiedliche Spielernummern („PlayerNo") auf jedem Endgerät. Unter Verwendung der Neutron Realtime Library geschieht die Zuordnung der Spielernummern automatisch.
Local Loopback dient zur Analyse der Techniken auf einem einzelnen Emulator oder Handy. In diesem Modus ist eine Simulation der QoS-Eigenschaften Latenz, Paketverlust und Jitter möglich. Die entsprechenden Parameter können im Netzwerk-Optionsmenü (siehe Abb. 3.11) eingestellt werden, das während der Simulation durch Druck auf die *GameD*-Taste (entspricht im WTK-Emulator der Taste „7") aufgerufen wird. Der Local-Loopback-Modus basiert intern genau wie der Peer-to-Peer-Modus auf dem DatagramManager, baut jedoch keine externe Verbindung auf. Stattdessen wird der Datagramm-Verkehr zunächst über die Latenz-Simulations-Komponente *LatencySimulator*, und schließlich über die IP-Loopback-Adresse (localhost) zurück an die Applikation geleitet, um die entfernte Darstellung der eigenen Entität nachbilden zu können. Die genaue Funktionsweise des LatencySimulators wird im Abschnitt 3.2.5 erläutert.

Optionsmenü zur Entitätserstellung

Nach Auswahl der gewünschten Netzwerkkommunikation erscheint das Optionsmenü zur Entitätserstellung, das auch später jederzeit durch Druck auf den linken Softkey aufgerufen werden kann. Hier lassen sich zunächst die grundsätzlichen Eigenschaften der lokalen Entity einstellen, die sich auf das Steuerungsverhalten auswirken.

Steuerungsverhalten der Entity:

acceleration: die maximale Linearbeschleunigung, die auf die Entity wirkt, sobald in eine Richtung beschleunigt wird.

drag: Mit drag wird die Wirkung des Gesamtwiderstands aller geschwindigkeitsabhängigen Reibungskräfte simuliert, die der Beschleunigung der Entity entgegenwirken. Der Drag legt indirekt die maximale Höchstgeschwindigkeit fest, da sich die beiden Kräfte ab einer bestimmten Geschwindigkeit ausgleichen.

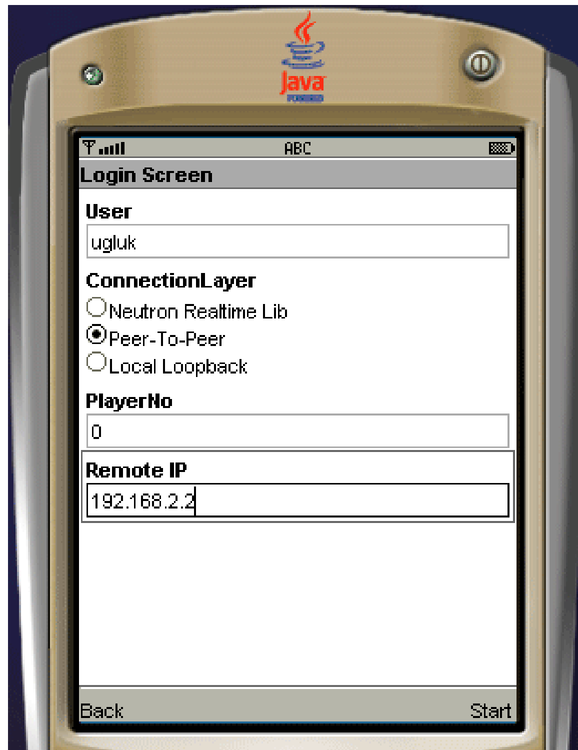


Abbildung 3.8: Startoptionen des Realtime Simulators

break force: Die Größe der automatischen Bremskraft, die auf die Entity wirkt, sobald nicht beschleunigt wird. Dieser Parameter ermöglicht es, das physikalisch vereinfachte Steuerungsverhalten einfacher Arcadespiele nachzubilden, in denen schlagartige Richtungswechsel möglich sind.

turn speed: Die lineare Winkelgeschwindigkeit beim Drehen einer Entity. Dieser Parameter ist nur von Bedeutung, falls *Thrust* oder *3D-Action* als SteeringModel gewählt wurde, da in der Direkten Steuerung die Orientierung schlagartig geändert wird.

Presets:

Typische Einstellungen verschiedener Genres wurden in vier Presets definiert, die sich über die Auswahlbox in der ersten Zeile anwählen lassen.

Sportsgame: Direkte Steuerung mit geringer Trägheit und entsprechend kurzen Beschleunigungsphasen; kein automatisches Bremsen.

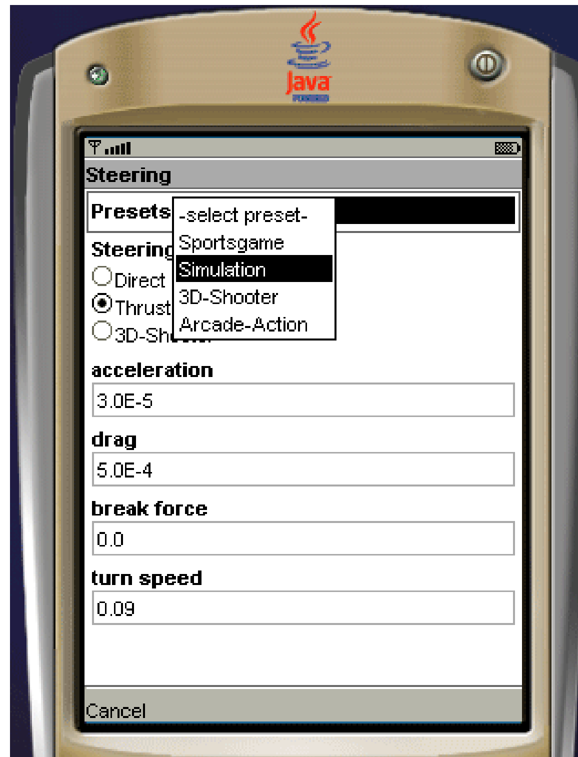


Abbildung 3.9: Optionseinstellungen zum Steuerungsverhalten der Entity

Simulation: Schuborientierte Steuerung mit hoher Trägheit und entsprechend langen Beschleunigungsphasen; kein automatisches Bremsen.

3D-Action: 3D-Action Steuerung mit geringer Trägheit; kein automatisches Bremsen.

Arcade-Action: Direkte Steuerung mit unrealistisch geringer Trägheit und extrem kurzen Beschleunigungsphasen; automatisches Bremsen.

Auf der zweiten Seite des Optionsmenüs werden die Einstellungen zum Darstellungsmodell und zugeordneten Konvergenzmodell gemäß Kapitel 3.2.3 vorgenommen. Nachdem die Einstellungen mit OK bestätigt wurden, wird eine neue Entity-Instanz mit den gewünschten Parametern lokal erstellt, der automatisch eine eindeutige ID zugewiesen wird. Gleichzeitig wird ein spezieller *CREATE_ENTITY*-Event an die Mitspieler gesendet. Nach dem Empfang wird bei den Mitspielern eine Entity mit den exakt gleichen Parametern erstellt, die sich nur im fehlenden Steuerungsmodell unterscheidet. Es ist nicht möglich, empfängerseitig ein anderes Darstellungsmodell als das des Senders zu benutzen, da sich Form und

Inhalt der übertragenen Daten je nach Darstellungsmodell unterscheiden. Das Erstellen und Interpretieren von Initialisierungs- und Update-Daten liegt dabei im Verantwortungsbereich der jeweiligen Darstellungs- und Konvergenzmodelle. Jedes Modell implementiert dazu die Methoden *createInitValues()*, *processInitValues(Hashtable)*, *createUpdateValues()* und *processUpdate(Hashtable)*.

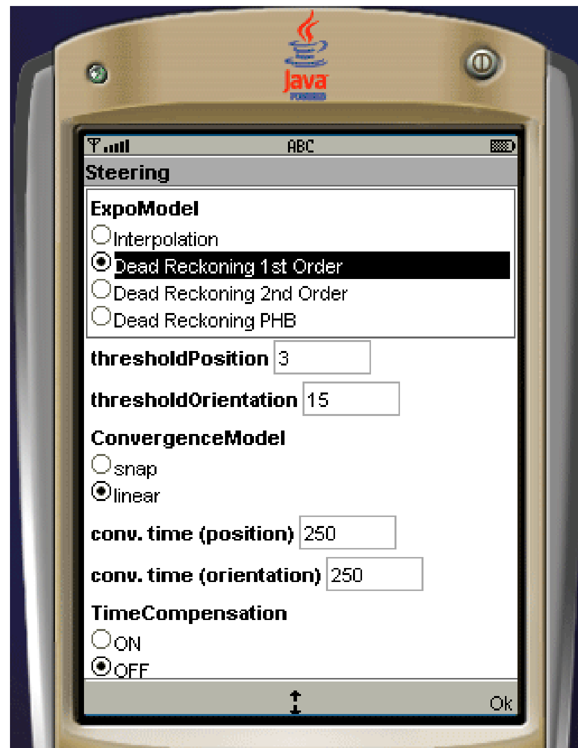


Abbildung 3.10: Optionseinstellungen des Darstellungs- und Konvergenzmodell der Entity

3.2.5 Simulation der Netzwerkeigenschaften

Um die Eigenschaften des Netzwerks zu simulieren, boten sich mehrere Möglichkeiten. Die erste bestand darin, spezielle Netzwerktreiber zu nutzen, die eigens für diesen Zweck entwickelt wurden. Es fanden sich jedoch nur kommerzielle Lösungen, die außerhalb unseres Budgets lagen. Außerdem konnten wir nicht sicher sein, dass eine solche Treiber-Lösung auch auf der lokalen IP-Loopback-Adresse funktionieren würden, was für unsere Testzwecke erforderlich gewesen wäre.
Die zweite Möglichkeit bestand darin, Algorithmen zur Simulation des Netzwerkverhaltens direkt in die Neutron Realtime Library einzubauen. Da wir die Neutron Realtime Library aufgrund der bereits erwähnten Instabilitäten aber umgehen wollten, entschieden wir uns für die

Implementierung der Simulationsalgorithmen in unserem *DatagramManager*. Abbildung 3.12 zeigt dabei beispielhaft den typischen Ablauf beim Versenden von Datagrammen: Zunächst werden die zu übermittelten Eventdaten per *sendData()*-Aufruf an den DatagramManager übergeben. Die Klasse *LatencySimulator* generiert für das jeweilige Event nun anhand der im Optionsmenü eingestellten Parameter eine individuelle Latenz (in Millisekunden), um die der Nachrichtenversand verzögert wird. Ist der Parameter *Packetloss* größer als Null, wird zunächst eine Zufallsfunktion ausgelöst, die den Event mit der gewünschten Wahrscheinlichkeit verwirft. In diesem Fall wird statt der Latenz der spezielle Returncode *RC_DROP_EVENT* zurückgegeben. Dieses Verhalten gibt die Realität allerdings nur vereinfacht wieder, da in der Neutron Realtime Library bei ausreichend hoher Senderate mehrere Events in einem Paket zusammengefasst werden. In diesem Falle ginge bei Paketverlust also nicht nur ein Event, sondern mehrere auf einen Schlag verloren. Für die Ermittlung erster Testwerte sollte diese Annäherung aber ausreichend sein, zumal wir in unseren Testläufen selten mehr als 5 Updates pro Sekunde auslösten.

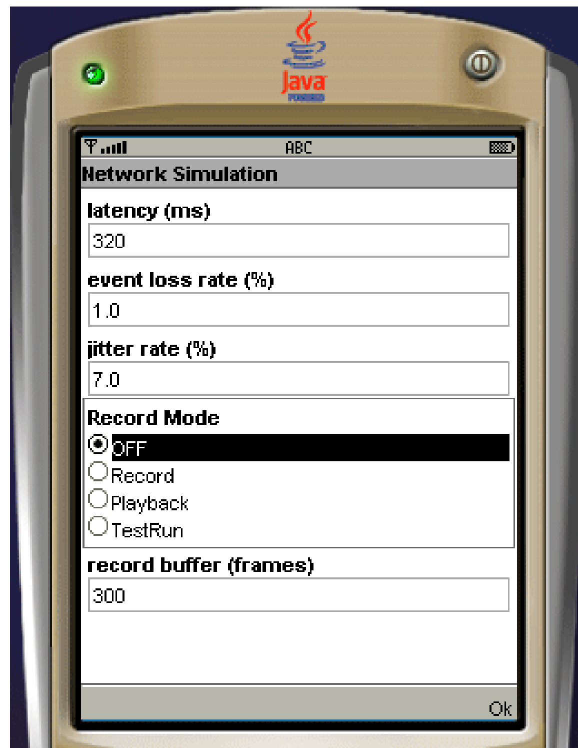


Abbildung 3.11: Optionsmenü zur Simulation der Netzwerk-Eigenschaften

Ein weiteres Problem unserer QoS-Simulation ist, dass wir auf dieser hohen Systemebene eine Begrenzung der Bandbreite nicht mit realistischem Aufwand simulieren können. Alternativ können wir aber die aktuell benötigte Bandbreite ermitteln, und auf diese Weise feststellen, welche Funknetze mit den aktuellen Optionseinstellungen überfordert sind. Dabei

ist zu beachten, dass sich das Transfervolumen noch durch die Header der unterliegenden Netzwerkschichten vergrößert.

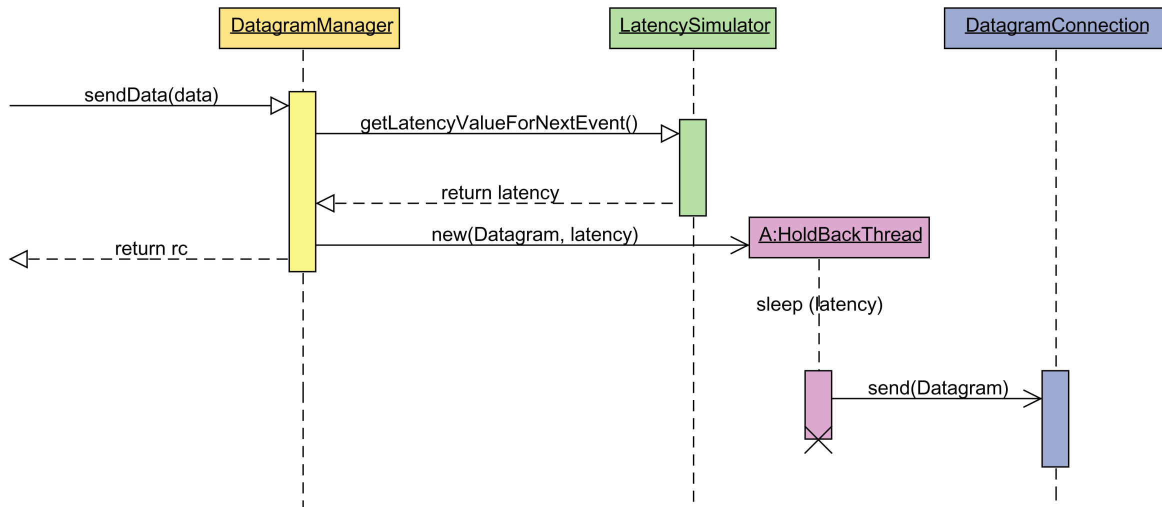


Abbildung 3.12: Sequenzdiagramm: Simulation der Netzwerkeigenschaften

Optionsmenü zur Netzwerk-Simulation:

Das Optionsmenü zur Einstellung der QoS-Parameter lässt sich im Local-Loopback-Modus nach Erstellung einer lokalen Entität durch Drücken der Taste „7“ (bzw. GameD) aufrufen.

latency: die minimal mögliche Latenz, die in jedem Fall nicht unterschritten wird

event loss rate: der Paketverlust in Prozent

jitter: die Latenz-Varianz in Prozent, abhängig vom aktuellen *latency*-Wert

Record Mode: Option zur statistischen Auswertung der aktuellen Einstellungen.

Der *Record Mode* dient zum Aufzeichnen typischer Bewegungsverläufe, um später die exakt gleichen Ausgangsdaten als Basis für die statistische Auswertung der verschiedenen Techniken nutzen zu können. In der Standard-Option *OFF* kann die Entität direkt gesteuert werden, und es findet keine Aufzeichnung statt. Im Modus *Record* werden alle Interaktionen in jedem Frame abgespeichert, zusammen mit der letzten Gameloop-Dauer in Millisekunden. Die Anzahl der aufzuzeichnenden Frames kann im Textfeld *recordbuffer* geändert werden. Im *Playback*-Modus werden die aufgezeichneten Daten einmalig abgespielt. Die Berechnung der Entitäten findet nun ausschließlich auf Basis der aufgezeichneten Interaktionen (je ein Bit-Array der gedrückten Tasten) und der aufgezeichneten Framedauer statt.

Während des Playbacks werden mehrere statistische Daten eingeblendet und ständig aktualisiert. Diese Daten werden im Abschnitt 3.3 erläutert. Die Statistiken lassen sich auch jederzeit im normalen Simulationsbetrieb (Record Mode = OFF) aktivieren. Der *TestRun*-Modus im Netzwerkmenü startet einen automatisierten Testlauf, der das aktuelle Playback immer wieder abspielt, wobei die Latenz und/oder die Paketverlust-Rate nach jedem Durchgang variiert werden. Die entsprechenden Daten sind als Arrays im Sourcecode definiert. Die jeweils aktuellen Einstellungen werden in der ersten Zeile der Statistik-Einblendungen angezeigt.

3.3 Auswertung der Techniken

Datenprotokoll der Algorithmen

In Abb. 3.13 ist die Größe der Updatepakete der verschiedenen Darstellungsmodelle aufgeführt. Dabei ist zu berücksichtigen, dass es sich nur um die reinen Nutzdaten handelt. Die Updatepakete vergrößern sich bei Verwendung der Neutron Realtime Library zusätzlich durch die UDP-Paket-Header und die Reliable-UDP-Funktionalität. Die Datentypen haben bei Java festgelegte Längen: ein Integer ist 4 Byte und ein Long ist 8 Byte lang. Demnach sind „PHB-DR" und „Interpolation" die Darstellungsmodelle mit den kleinsten Updatepaketen. Da aber ihre Updatefrequenz unter Umständen wesentlich höher sein kann, lässt sich eine Aussage über das eigentliche Datenaufkommen der Darstellungsmodelle erst durch die Auswertung der Tests machen.

	DIS-DR 1. Ord.	DIS-DR 2. Ord.	PHB-DR	Interpolation
Id (1 Byte)	X	X	X	X
Position (2 Int)	X	X	X	X
Orientation (1 Int)	X	X	X	X
Linear Velocity (2 Int)	X	X		
Linear Acceleration (2 Int)		X		
Angular Velocity (1 Int)	X	X		
Timestamp (1 Long)	with time compensation	with time compensation	X	X
Total Bytes	25 / 33	33 / 41	21	21

Abbildung 3.13: Nutzdaten der Darstellungsmodelle

Rechenaufwand der Algorithmen

Ein weiterer Aspekt, der bei der Wahl eines geeigneten Algorithmus' für ein Spielgenre beachtet werden muss, ist der benötigte Rechenaufwand. Es kann sein, dass der vermeintlich optimale Algorithmus für ein Spielgenre ab einer bestimmten Anzahl zu verwaltender Entitäten, die Handys überfordert. Ist dies der Fall, muss auf einen Algorithmus ausgewichen

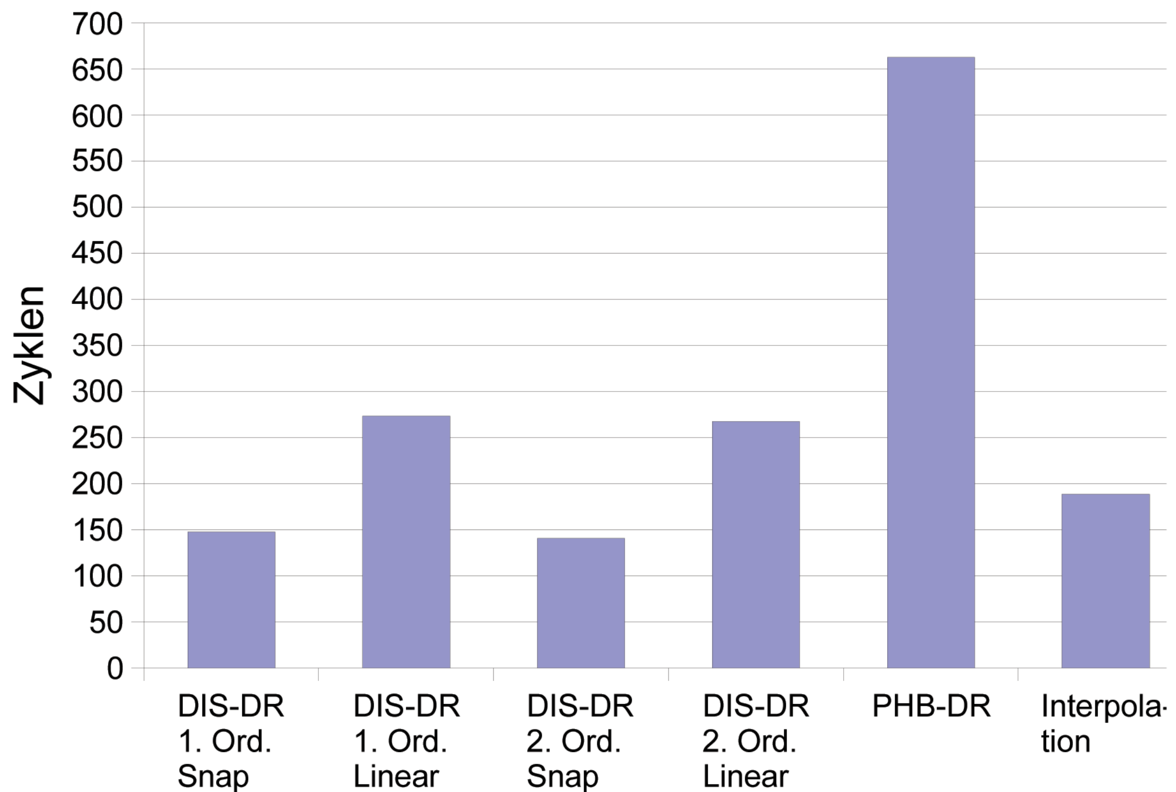
werden, der das beste Verhältnis zwischen Spielbarkeit und Rechenaufwand bietet.
Um den Rechenaufwand der von uns implementierten Algorithmen festzuhalten wurde der WTK-Profiler verwendet (*WTK (Wireless Tool Kit): Ein von Sun angebotenes Paket, welches die Emulatoren und API's bereitstellt, die für die Progammierung von J2ME-Handy-Applikationen benötigt werden. Profiler: eine Funktion der WTK-Emulatoren, die die anteilige Rechenzeit aller Methoden des ausgeführten Programms protokolliert*). Um eine genaue Auswertung dieser Daten zu ermöglichen, wurde ein Testprogramm geschrieben, dessen einzige Funktion es ist alle Algorithmen aufgrund eines Updates auszuführen. Hierbei wurden Fehlerquellen, wie der Gebrauch von rechenintensiven Hilfsmethoden die nicht Teil des eigentlichen Algorithmus sind und daher die Erhebung des Rechenaufwands verfälschen könnten, ausgeschlossen. Trotzdessen sind die ermittelten Ergebnisse nur im geschlossenen Umfeld dieser Arbeit repräsentativ, da der Rechenaufwand einzelner Algorithmen, durch eine Optimierung der Implementierung oder der Verwendung einer Hardwarenäheren Programmiersprache, verbessert werden könnte. Die unter Abb. 3.3 aufgeführten Messdaten sind die Mittelwerte aus fünf Testläufen und geben die Anzahl der Rechenzyklen an, die jeder Algorithmus anteilig benötigte. Wie erwartet zeigt sich, dass „PHB-DR“ der rechenintensivste Algorithmus ist. Bei „PHB-DR“ wird ca. 4 mal soviel Rechenaufwand betrieben wie bei „DIS-DR mit Snap-Konvergenz“, dem genügsamsten Algorithmus. „Interpolation“ liegt mit durchschnittlich 189 Zyklen zwischen den beiden „DIS-DR“-Konvergenzalgorithmen. Lineare Konvergenz erzeugt mit durchschnittlich 270 Zyklen den meisten Rechenaufwand der „DIS-DR“-Konvergenzalgorithmen, steht aber immer noch 2,5 mal besser da als „PHB-DR“. Die Berücksichtigung der Beschleunigung bei „DIS-DR zweiter Ordnung“ fällt im Vergleich zu „DIS-DR erster Ordnung“, weder bei Snap noch bei Linearer Konvergenz ins Gewicht. Abschließend lässt sich sagen, dass die in Bezug auf den Rechenaufwand günstigsten Algorithmen „DIS-DR erster oder zweiter Ordnung mit Snap-Konvergenz“ oder „Interpolation“ sind. Spielegenres für die möglicherweise ein anderer Algorithmus die beste Lösung in Bezug auf Konsistenz und Spielbarkeit darstellt, sollten also bei zu hoher Prozessorbelastung einen dieser drei Algorithmen als Alternative in Betracht ziehen.

Testaufbau

Im folgenden Teil nutzen wir den TestRun-Modus, um für jedes Spielgenre das geeignetste Darstellungsmodell zu finden. Die Tauglichkeit einer Technik beurteilen wir nach folgenden Kriterien:

Durchschnittliche Senderate (avrgDataSent)

Die durchschnittlich benötigte Bandbreite der ausgehenden Updates, die in kilo-Bytes pro Sekunde gemessen wird. Hierbei ist zu beachten, dass dieser Wert in der Praxis noch durch Paketheader erhöht wird. Wie stark dieser Wert von der Realität abweicht, hängt von den



Eigenschaften des unterliegenden Kommunikationsprotokolls ab, da z. B. in der Neutron Realtime Library aus Performancegründen mehrere Events in einem Paket zusammengefasst werden. Die Betrachtung der reinen Nutzdaten lässt einen unverfälschten Vergleich der verwendeten Techniken zu.

Maximale Senderate (maxDataSent)

Die maximale Bandbreite, die im aktuellen Testlauf bisher benötigt wurde. Hier ist es wichtig zu vergleichen wie das Verhältnis zur durchschnittlichen Senderate, für die verschiedenen Techniken, ausfällt. Durch einen Vergleich mit Tabelle 2.2 lassen sich Aussagen über die Realisierbarkeit der Spielegenres im Hinblick auf die QoS-Eigenschaften der Mobilfunk-Technologien treffen.

Durchschnittlicher Positionsfehler (avrgDistanceError)

Die durchschnittliche Abweichung in Pixeln, der lokal dargestellten entfernten Entität zur tatsächlichen momentanen Position. Anhand dieses Wertes lässt sich ablesen, wie tauglich die momentane Technik für Spiele mit Projektilwaffen ist, und wie empfindlich sich Kollisionen auf den Spielfluss auswirken werden. Als Richtwert dient die Länge des Dreiecks unserer Standard-Entity, die 10 Pixel beträgt. Eine durchschnittliche Abweichung von 10 Pixeln würde also bedeuten, dass ein Projektilschuss, der das Objekt in der lokalen Darstellung genau in der Mitte trifft, das Objekt in 75% aller Fälle trotzdem verfehlt, sofern die Treffer-Auswertung auf einem entfernten System erfolgt und sich das Objekt rechtwinklig zur Schussrichtung bewegt. Nach dem gleichen Prinzip können auch Kollisionen mit anderen Spielobjekten zu Irritationen führen, die serverseitig erkannt werden, obwohl der Spieler in seiner Darstellung erfolgreich ausgewichen ist. Der durchschnittliche Positionsfehler gibt dann die Pixelanzahl an, um die die Entität bei Kollisionen wahrscheinlich neu positioniert wird.

Maximaler Positionsfehler (maxDistanceError)

Der maximale Positionsfehler, der im aktuellen TestRun bisher aufgetreten ist. Wie verhält er sich zum durchschnittlichen Positionsfehler der den verschiedenen Techniken?

Bewegungsverlauf

Dieser Wert beschreibt, wie originalgetreu der ursprüngliche Bewegungsverlauf von der aktuellen Technik wiedergegeben wurde. Folgt die Entität dem ursprünglichen Bewegungsverlauf? Zittert die Darstellung? Gibt es Sprünge in der Animation? Dieses Verhalten hängt von mehreren Faktoren ab und ist schwierig in Zahlen zu fassen, deshalb haben wir Schulnoten von 1 (sehr gut) bis 6 (ungenügend) vergeben.

3.3.1 Sportspiel

Dieses Steuerungsmodell ist realistischen Sportspielen wie der EA FIFA-Serie oder Pro Evolution Soccer nachempfunden. Es wird das Direkte Steuerungsmodell unterstützt, und die Entität beschleunigt sehr schnell. Trotzdem wird ein realistisches Physikmodell genutzt, und es ist eine leichte Trägheit spürbar. Ein Richtungswechsel um 180° aus vollem Lauf nimmt in etwa eine Sekunde in Anspruch.

Bewegungsverlauf

Um eine Vorauswahl treffen zu können, betrachten wir zunächst, wie gut die verschiedenen Modelle den Bewegungsverlauf wiedergeben. Hier zeigte sich schnell, dass Time Compensation Techniken mit diesem Genre Probleme haben. Time Compensation Techniken basie-

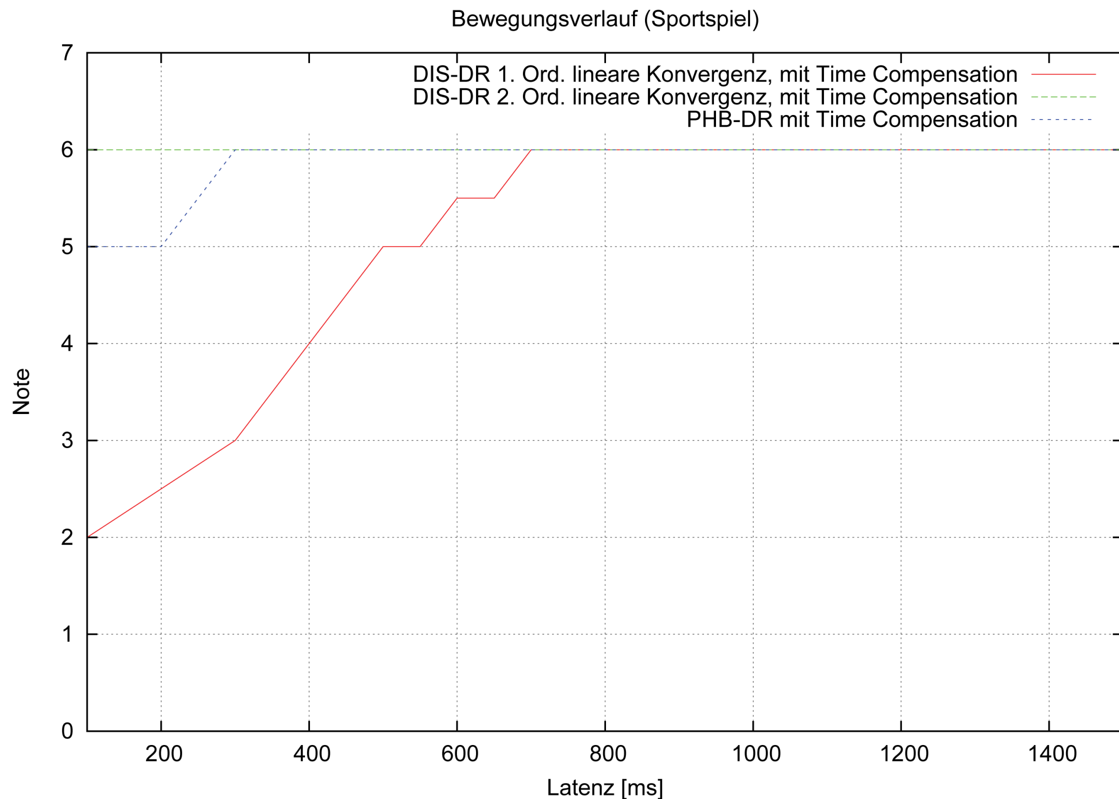


Abbildung 3.14: Bewegungsverlauf, Verfahren mit Time Compensation (Sportspiel)

ren darauf, dass ein punktuell gemessener Zustand über die Dauer einer ganzen Roundtrip-Time (RTT) gilt. Das ist in Sportspielen aufgrund der relativ abrupten Richtungswechsel selten der Fall, weshalb der berechnete Konvergenzpunkt oft stark von der tatsächlichen Position abweicht. Die relativ hohen Latenzen im Bereich der Mobilfunknetze verstärken den Positionsfehler zusätzlich. Die Folge sind Zickzack-Bewegungen (bei linearer Konvergenz) bzw. Sprünge (bei Snap-Konvergenz) quer zum tatsächlichen Bewegungspfad, die mit steigender Latenz immer extremer werden. Am katastrophalsten schneidet DIS-DR 2. Ordnung ab, da hier von einer konstanten Beschleunigung für die Dauer einer RTT ausgegangen wird, wobei diese Beschleunigung in Wirklichkeit nur für wenige Millisekunden anliegt. Diese Technik führt schon bei einer Latenz von 100 ms zu unzumutbaren Fehlern (siehe Abb. 3.14). Kleinstes Übel unter den Time Compensation Algorithmen ist DIS-DR 1.Ordnung, das bei einer Latenz von 400 ms noch halbwegs zumutbare Werte liefert. Damit kommt diese Technik für dieses Genre prinzipiell noch für mobile UMTS-Geräte in Frage, scheidet aber für GPRS-Netze aus.

Bei Darstellungsmodellen ohne Time Compensation bleibt die Authentizität des Bewegungsverlaufs auch mit steigender Latenz durchgehend konstant, da die Entität exakt die gleichen

Bewegungen ausführt, nur mit immer größerem Zeitversatz. Beim Vergleich der Bewegungsverläufe genügen also einzelne Werte, die in Abb. 3.15 gegenübergestellt werden.

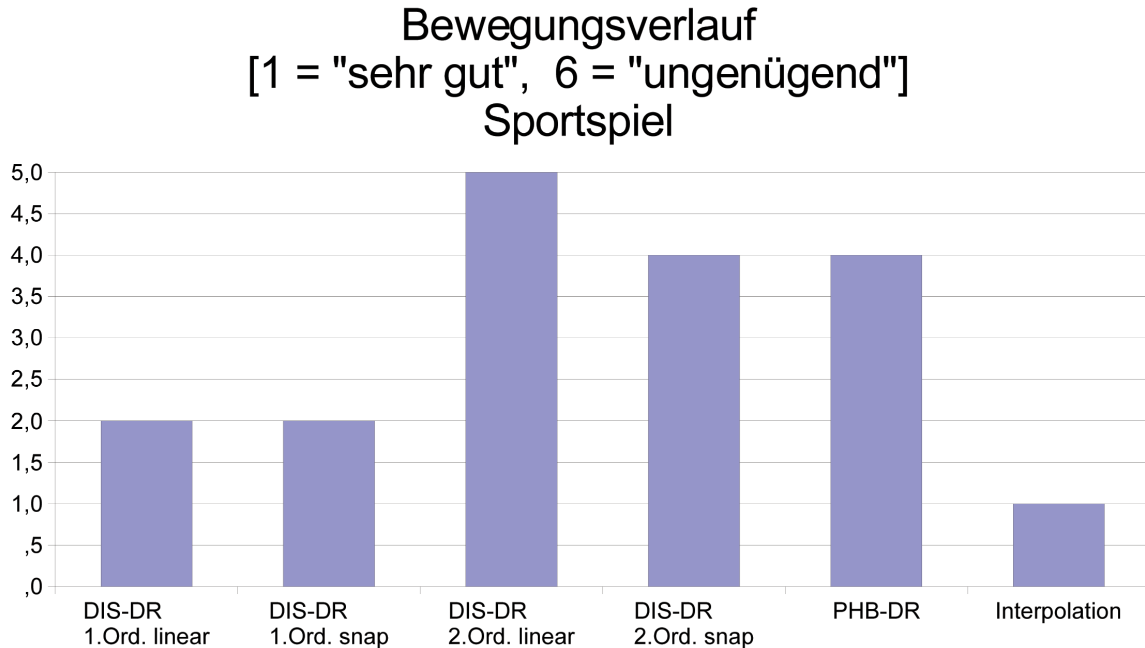


Abbildung 3.15: Bewegungsverlauf, Verfahren ohne Time Compensation (Sportspiel)

Hier schneidet Interpolation mit „sehr gut“ am besten ab, dicht gefolgt von DIS-DR 1. Ordnung mit „gut“. Hierbei machte ein Umschalten auf Snap-Konvergenz erstaunlich wenig Unterschied, da die Konvergenzsprünge nur sehr klein und meist gar nicht sichtbar waren. Bei DIS-DR. 2. Ordnung führte die lineare Konvergenz (mit 250 ms Konvergenzzeit) aufgrund des fehlerhaft berechneten Konvergenzpunktes zu relativ starken Zickzack-Bewegungen quer zur tatsächlichen Bewegungsrichtung. Snap-Konvergenz wurde als „ausreichend“ bewertet, da der Bewegungsverlauf besser eingehalten wurde. Dafür mussten gelegentliche Sprünge über mehrere Pixel in Kauf genommen werden, die wir aber als weniger störend empfunden haben. Durch eine Verringerung der Konvergenzzeit ließe sich hier ein Kompromiss erzielen. PHB-DR hatte mit einer leicht stotternden Bewegung zu kämpfen. Noch störender aber war hier ein ständiges Rotieren der Entität, das dadurch entsteht, dass PHB-DR aus den letzten Orientierungswinkeln eine Winkelgeschwindigkeit errechnet. Das schlagartige Wechseln der Orientierung führt dazu, dass die Entität ständig über den tatsächlichen Winkel hinaus rotiert. Dieses Problem trat auch bei den DIS-DR Algorithmen auf. Allgemein ist beim Direkten Steuerungsmodell anzuraten, auf das Extrapolieren der Winkelgeschwindigkeiten zu verzichten, und stattdessen die Orientierung einfach auf die des letzten Updates zu setzen.

Datenaufkommen

Der originalgetreue Bewegungspfad der DIS-DR Algorithmen erklärt sich dadurch, dass durch das schlagartige Ändern der Orientierung bei jedem Tastendruck der Threshold überschritten wird, und entsprechend viele Updates gesendet werden. Das führt zu einer entsprechend hohen Datentransferrate, die mit 0,43 kB/s um ca. 35% höher liegt als die der Interpolation, die mit 0,28 kB/s die geringste Netzwerklast verursacht (siehe Abb. 3.16 , 3.17). Zusätzlicher Vorteil der Interpolation ist der kaum vorhandene Unterschied zwischen maximalem und durchschnittlichen Datenaufkommen pro Sekunde (Abb. 3.17). In hektischen Spielsituationen stieg das Datenaufkommen pro Spieler bei Dead Reckoning Modellen dagegen auf bis zu 0,85 kB/s (PBH-DR), was bei einer ausreichend hohen Spieleranzahl und geringer Bandbreite zu Engpässen und dadurch zu Rucklern im Spielverlauf führen kann.

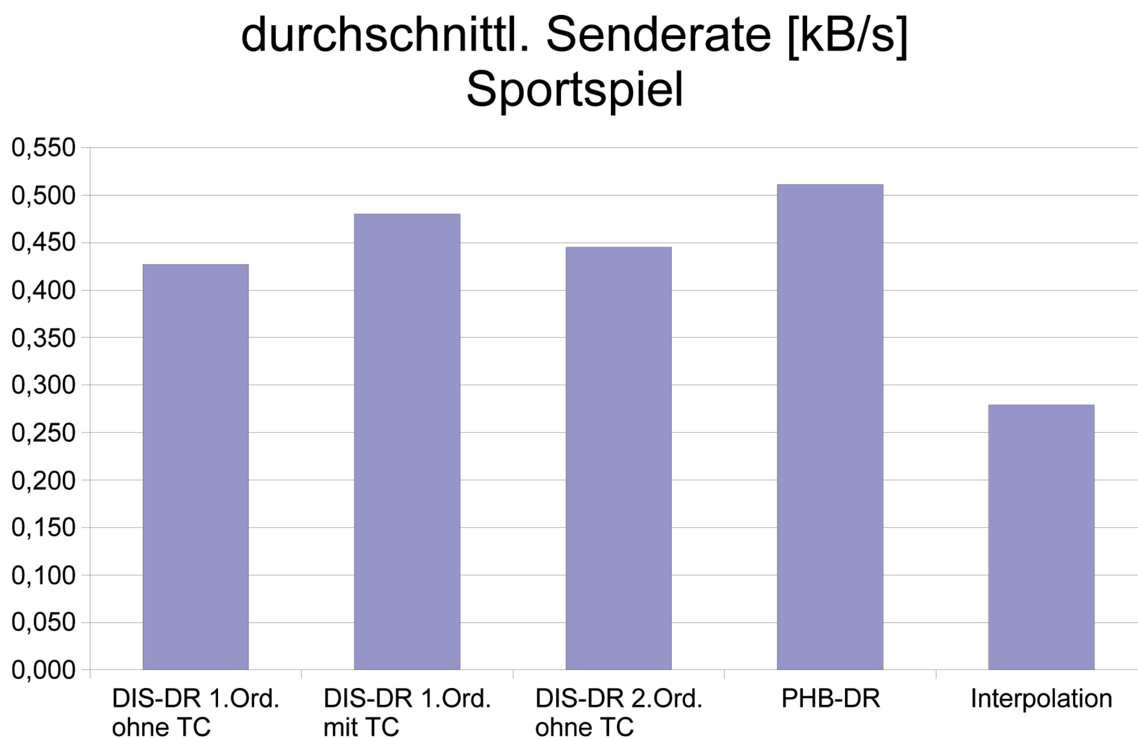


Abbildung 3.16: durchschnittliche Senderate (Sportspiel)

Positionsfehler

Der große Nachteil der Interpolation ist die Abweichung der dargestellten Position zur tatsächlichen momentanen Position der entfernten Entität. In unseren Testläufen wird mit steigender Latenz automatisch die Deltazeit angepasst, indem 300 ms zur aktuellen Latenz

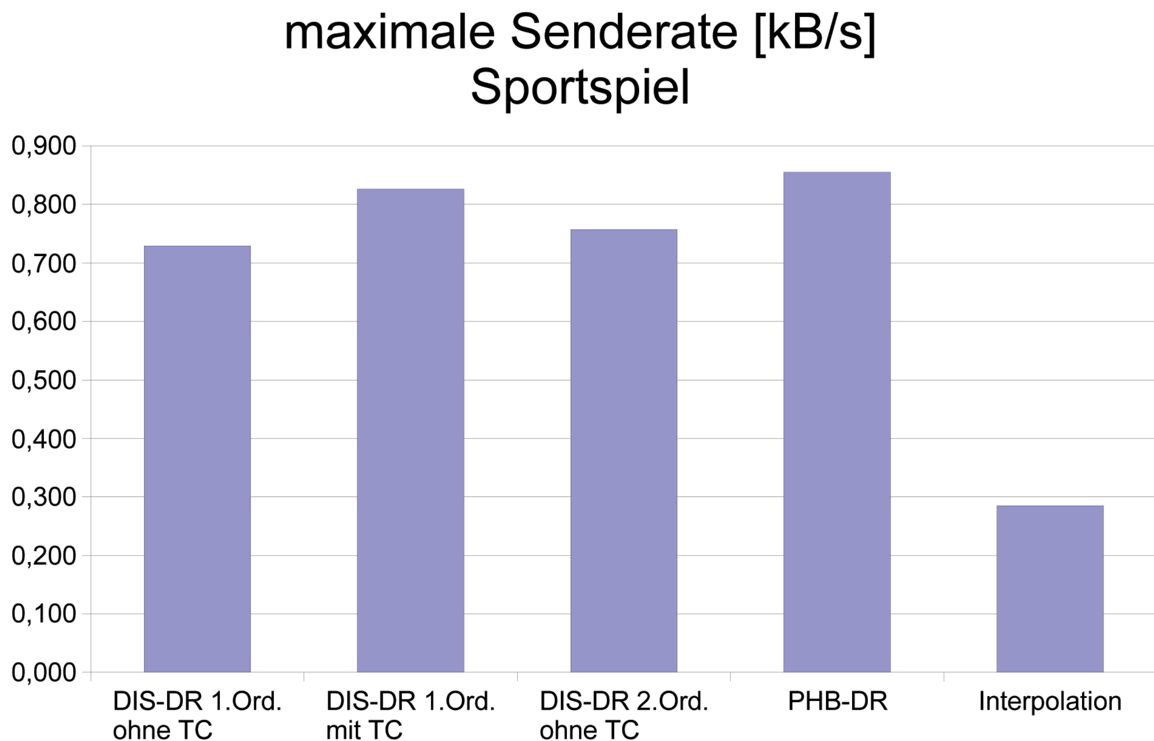


Abbildung 3.17: maximale Senderate (Sportspiel)

hinzu addiert wurden. In Bezug auf das konstante Updateintervall von 250 ms und die Gameloopdauer von etwa 40 ms wurde die Deltazeit damit auf den kleinstmöglichen Wert gesetzt. Da im ersten Testlauf kein Paketverlust simuliert wurde, werden alle Pakete rechtzeitig verarbeitet, und die Entität gerät nie ins Stocken. Entscheidet man sich in der Praxis dafür, die Deltazeit so hoch zu setzen, dass der Verlust einzelner Pakete ohne Stocken der Bewegung toleriert werden kann, so gelten für den jeweiligen Latenzwert die Werte, die im gleichen Diagramm an der Position des doppelten Latenzwertes notiert sind. Ohne Berücksichtigung von Paketverlust beträgt der Positionsfehler bei Interpolation bei 400 ms RTT bereits 22 Pixel. Plant man Paketverlust ein, steigt der Fehler auf 33 Pixel. Auch im optimalen Fall bringt die Interpolation unter allen Modellen die größte durchschnittliche Abweichung mit sich (Abb. 3.18). DIS-DR 1.Ordnung schneidet hier am besten ab. Dieser scheinbare Vorteil relativiert sich jedoch, wenn wir in Abb. 3.19 den maximalen Positionsfehler einer Testrunde betrachten: Ab einer Latenz von 800 ms liefert dieses Modell den schlechtesten Wert.

Sportspiele unter GPRS

Die Auswertungen haben gezeigt, dass sich Sportspiele nach Vorbild der FIFA- oder Pro Evolution Soccer-Serie unter GPRS auf keinen Fall realisieren lassen. Auch ohne die Be-

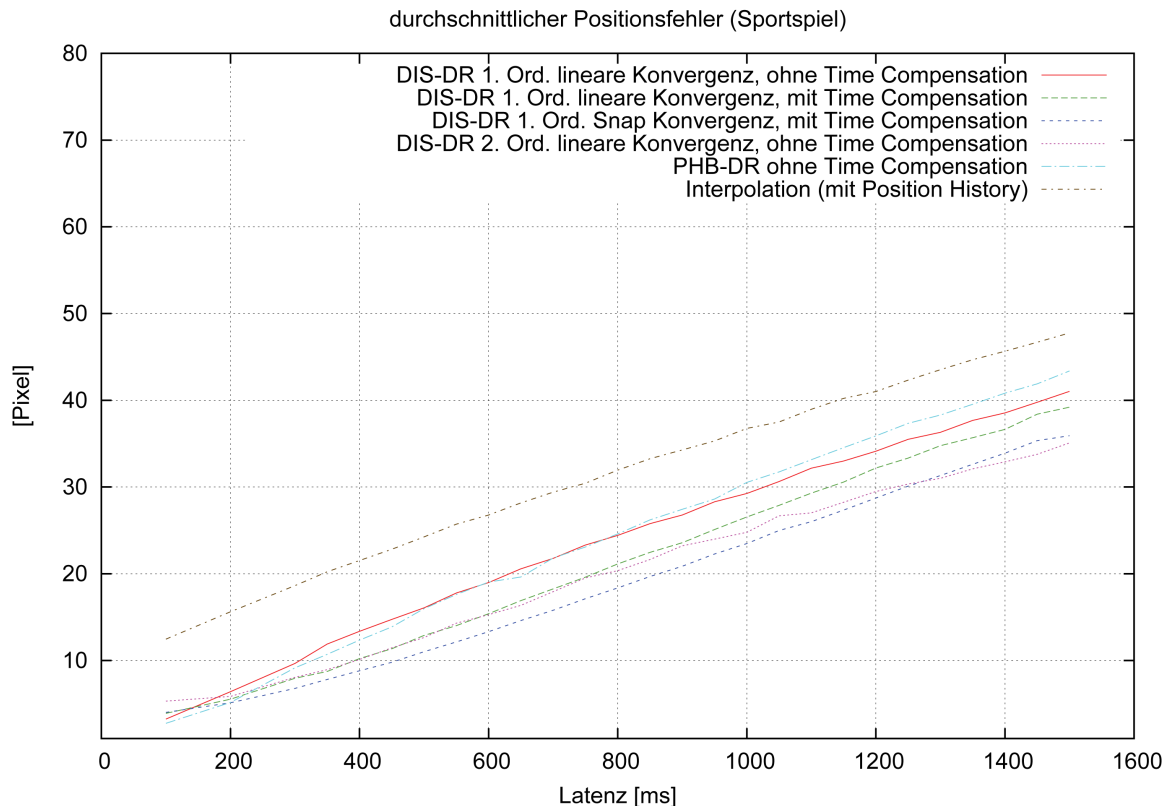


Abbildung 3.18: durchschnittlicher Positionsfehler (Sportspiel)

rücksichtung von Paketverlust und Jitter ist eine Latenz von 800 ms bereits so hoch, dass die Positionsfehler ein vernünftiges Spielen unmöglich machen. Auch wenn starke Abstriche im Bewegungsverlauf in Kauf genommen werden, werden Kollisionsabfragen bei derart hohen Werten zur Glückssache. Ein hoher Jitter würde zudem dazu führen, dass ein exaktes Timing für Torschüsse und Passspiel unmöglich gemacht wird, da der Spieler nicht einkalkulieren kann, wann die Bestätigung der Gegenseite ankommt und der Ball den Fuß der Spielfigur verlässt.

Sportspiele unter UMTS

Unsere Testläufe haben gezeigt, dass verteilte Sportspiele unter UMTS wahrscheinlich zufriedenstellend realisiert werden können. Mit Dead Reckoning 1. Ordnung und aktivierter Time Compensation existiert ein Modell, das den durchschnittlichen Positionsfehler in einem akzeptablen Bereich hält, und gleichzeitig den Bewegungsverlauf gut wiedergibt. Für die Feinjustierung der Parameter unter UMTS gingen wir von den in 2.4.5 aufgeführten Durchschnittswerten aus. Wie empfehlen dazu folgende Einstellungen:

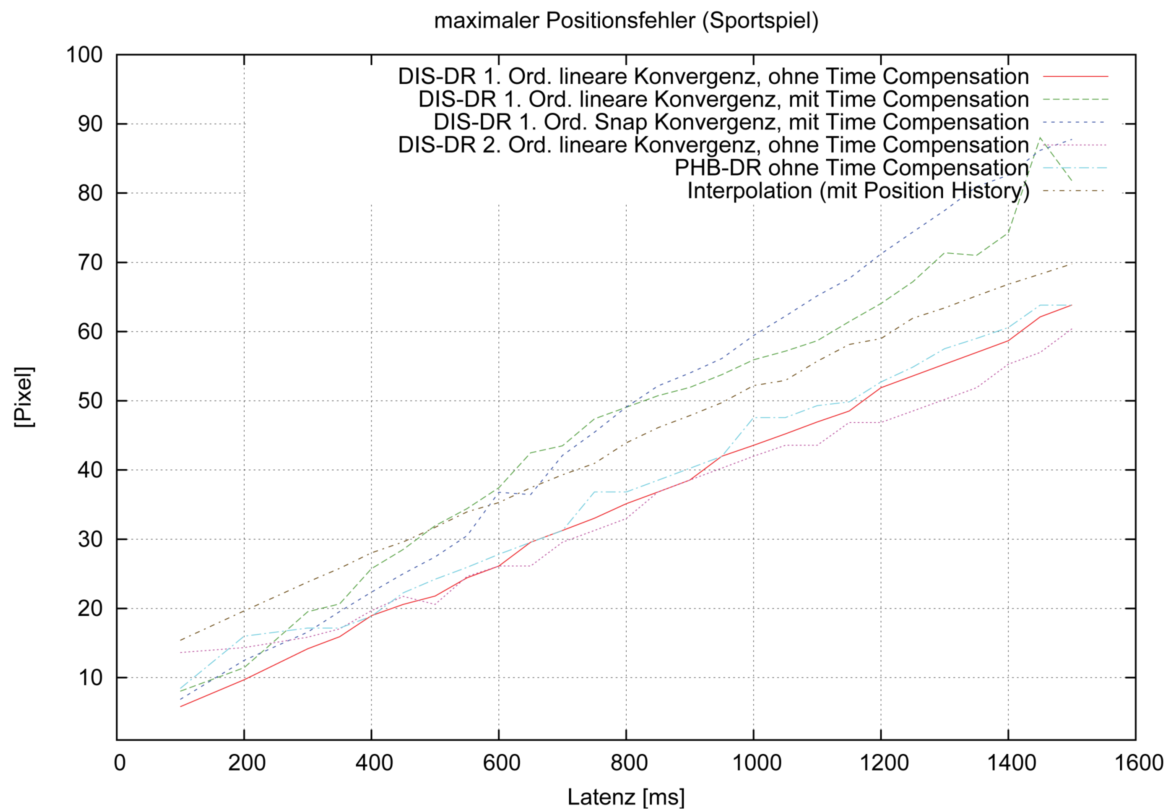


Abbildung 3.19: maximaler Positionsfehler (Sportspiel)

Darstellungsmodell:	DIS-DR 1.Ordnung mit Time Compensation, begrenzt auf 200 ms
Threshold Position:	10% der Größe des Spielobjekts (1 Pixel)
Threshold Orientierung:	unerheblich, muss nur < 45° sein
Konvergenzmodell:	Lineare Konvergenz mit jeweils 100 ms Konvergenzeit

Mit diesen Einstellungen wurden folgende Werte erzielt:

Bewegungsverlauf:	gut (2.0)
durchschnittl. Senderate (Nutzdaten):	0,523 kByte/s
maximale Senderate (Nutzdaten):	0,826 kByte/s
durchschnittl. Positionsfehler:	4,5 Pixel (0,45 % der Größe des Spielobjekts)
maximaler Positionsfehler: 9,78 Pixel	(0,98 % der Größe des Spielobjekts)

In Fußballspielen muss oft sehr genau gegrätscht oder exakt zum Ball gelaufen werden werden, deshalb stellt der maximale Positionsfehler den kritischsten Parameter dar. Die entfernte Darstellung wich unter den empfohlenen Einstellungen noch maximal um etwa die Länge

des Spielobjekts ab. Anhand unserer Testapplikation ist schwer zu sagen, ob dieser Wert in der Praxis noch toleriert werden kann. Um den Positionsfehler weiter zu verringern, müsste entweder die Steuerung noch träger gemacht, oder die Geschwindigkeit des Spielobjekts verringert werden.

3.3.2 Simulationen

Beim Testlauf des Simulations-Presets untersuchten wir Spielegenres, deren Entitäten versuchen, ein genaueres Abbild realer Vehikel zu simulieren. Rennwagen oder Raumschiffe weisen meistens eine weitaus niedrigere lineare Beschleunigung und höhere Trägheit auf, als die Entitäten der anderen Genres.

Bewegungsverlauf

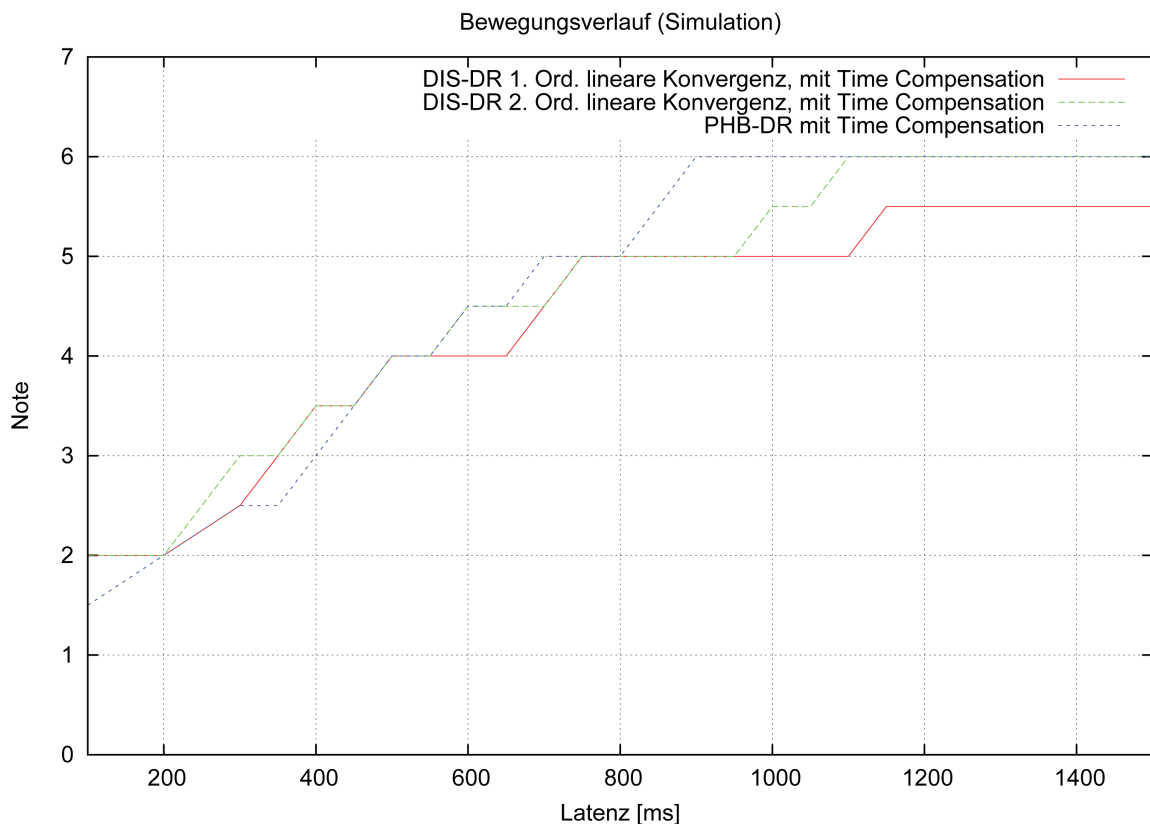


Abbildung 3.20: Authentizität des Bewegungsverlaufs bei Simulationen (Techniken mit Time Compensation)

Beim trägen Bewegungsmodell des Simulations-Presets schneiden die Techniken mit Time

Compensation deutlich besser ab als im Sportspiel-Preset. Die Dead Reckoning Techniken liegen in etwa gleich auf, und liefern noch bis ca. 350 ms Latenz zufriedenstellende Ergebnisse (siehe Abb. 3.20). Ab Latenzen über 1000 ms wird der Zickzack-Kurs quer zur Bewegungsrichtung als unzumutbar bewertet, wobei DIS-DR 1.Ordnung noch einen Tick erträglicher bleibt. Am störendsten wirkt sich bei den Dead Reckoning Modellen die abweichende Orientierung aus, die mit steigender Latenz größere Fehler als die Positionsvorhersage mit sich bringt. Dieser Ansatz eignet sich nicht gut für das Steuerungsmodell von Simulationen, das eine feste Drehgeschwindigkeit ohne Beschleunigung umfasst. Durch Anpassungen am Algorithmus (kein Extrapolieren des Rotationswinkels, sondern Interpolieren) würde dieser Fehler abgeschwächt, sodass auch bei Latenzen um die 600 ms noch zufriedenstellende Ergebnisse erzielt werden könnten.

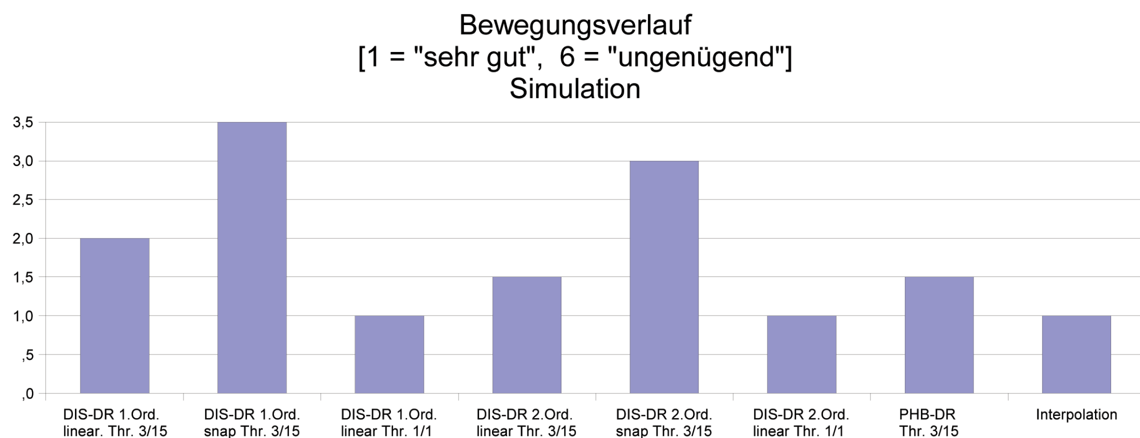


Abbildung 3.21: Authentizität des Bewegungsverlaufs bei Simulationen, Techniken ohne Time Compensation

Darstellungsmodelle ohne Time Compensation (siehe Abb. 3.21) schneiden insgesamt gut bis sehr gut ab, wobei die Snap-Konvergenz im Gegensatz zum Sportspiel-Preset nun deutliche Sprünge mit sich bringt. Hintergrund ist der Threshold, der nun nicht mehr bei jedem Tastendruck überschritten wird, weshalb wir auch Testläufe mit unterschiedlichen Schwellwerten durchgeführt haben. Als praktikabler Kompromiss zwischen Updaterate und originalgetreuem Bewegungsverlauf hat sich hier ein Schwellwert von 3 Pixeln Positionsabstand und einer Winkeldifferenz von 15° bewährt. Ein optimaler Bewegungsverlauf wird bei einem Threshold von 1 Pixel und 1° erzielt, der sowohl bei DIS-DR 1. Ordnung als auch 2. Ordnung zu einer Bewertung von „sehr gut“ geführt hat. Bei der Interpolation wurde das Updateintervall von 250 ms beibehalten, was auch hier zu einer sehr guten Wiedergabe des Bewegungsverlaufs geführt hat. Wer die Netzwerklast weiter senken will, erreicht auch bei einem Updateintervall von 500 ms noch zufriedenstellende Ergebnisse. Position History Based Dead Reckoning schnitt bei Schwellwerten von 3 Pixeln und 15° ebenfalls nahezu perfekt ab. Es waren nur

gelegentlich leichte Unregelmäßigkeiten im Bewegungsverlauf erkennbar, deshalb die Note 1,5.

Datenaufkommen

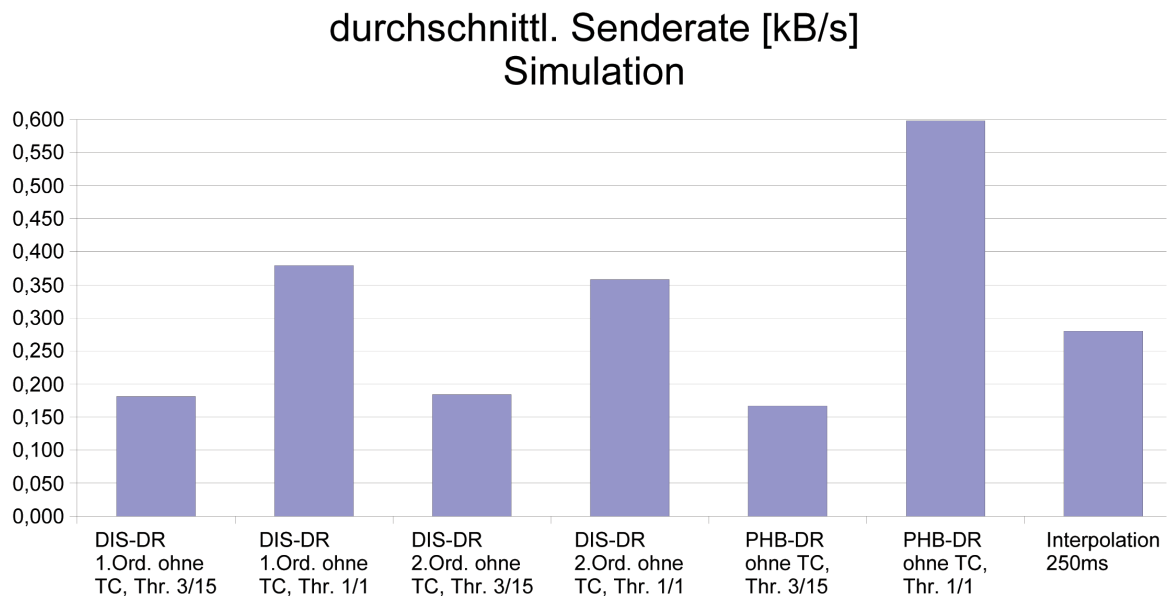


Abbildung 3.22: durchschnittliches Datenaufkommen pro Spielobjekt (Simulation)

Beim Blick auf das verursachte Datenaufkommen in Abb. 3.22 und 3.23 zeigt sich, dass PHB-DR mit einer durchschnittlichen Transferrate von 0,167 kB/s den niedrigsten Wert erzielte, wobei auch die maximale Rate nie über 0,285 kB/s stieg. Die beiden DIS-DR Techniken schnitten nur unwesentlich schlechter ab. Das Minimieren der Schwellwerte führte jeweils annähernd zu einer Verdoppelung des durchschnittlichen und des maximalen Datenaufkommens, wobei sich dieser Wert bei PHB-DR sogar in etwa verdreifachte. Das Aktivieren von Time Compensation erhöhte die Datentransferrate bei den DIS-DR Techniken nur um etwa 5%. Bei PHB-DR ergibt sich keine Änderung, da dieses Modell auch ohne Time Compensation mit Zeitstempeln arbeitet, und daher keine zusätzlichen Daten übertragen werden müssen. Interpolation bietet bei Simulationen im Hinblick auf die Netzlast keinerlei Vorteile, da das Datenaufkommen nun auch bei den Dead Reckoning Techniken im Vergleich zum Sportspiel-Preset nicht nur wesentlich niedriger, sondern auch wesentlich konstanter ist.

Positionsfehler

Im Hinblick auf den Positionsfehler zeigt sich der große Vorteil der Nutzung von Time Compensation: Mit DIS-DR 1. Ordnung bleibt der durchschnittliche Fehler auch bei einer Latenz

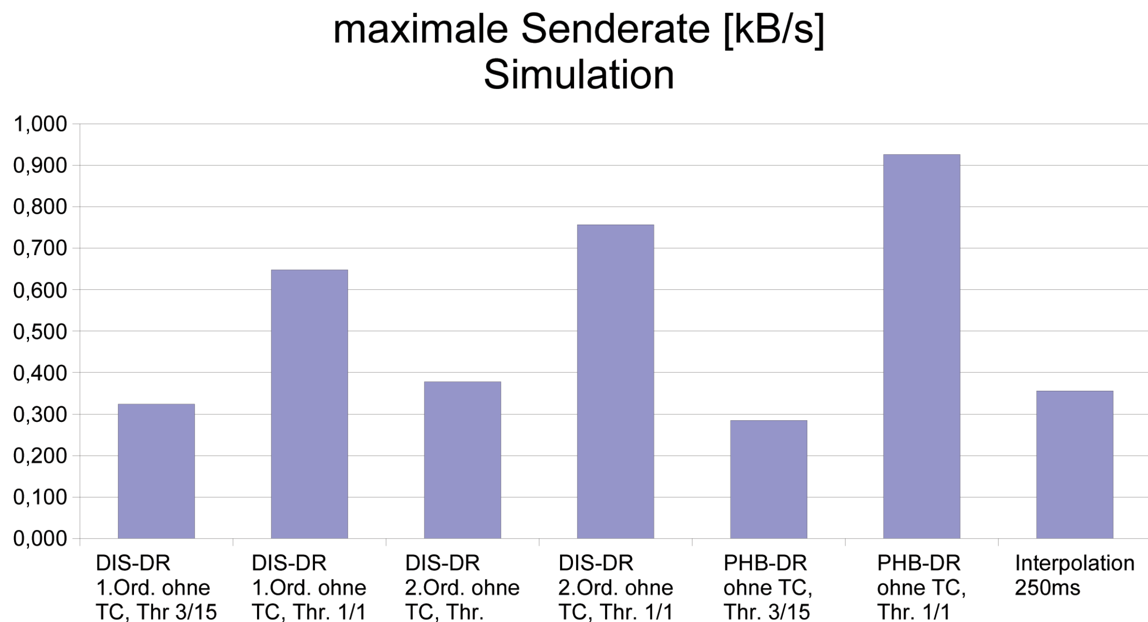


Abbildung 3.23: maximales Datenaufkommen pro Spielobjekt (Simulation)

von 800 ms nur knapp über der Größe des Spielobjekts, und beträgt dabei maximal dessen 2 1/4 fache Länge. Auch PHB-DR liefert sehr gute Ergebnisse, wobei die Werte je nach Latenz etwas stärker variieren. Die Interpolation liefert auch hier wieder die größte Abweichung.

Simulationen unter GPRS

Simulationen und Rennspiele sind unter GPRS nur bedingt umsetzbar. Interpolation ist hier aufgrund des hohen durchschnittlichen Positionsfehlers ungeeignet. Bei Dead Reckoning 1. Ordnung war mit 600 ms die höchste maximale Time Compensation möglich, ohne den Bewegungsverlauf zu stark zu verfälschen. Dieses Modell war gleichzeitig auch am tolerantesten gegenüber Paketverlust, der bei einer Quote von 5% kaum erkennbar war. PHB-DR reagierte hier extrem empfindlich und führte zu Sprüngen von bis zu 100 Pixeln. DIS-DR 2. Ordnung benötigte zwar die schmalste Bandbreite, bot aber einen sehr unsauberen Bewegungsverlauf, und führte bei Paketverlust zu größeren Positionsfehlern als DIS-DR 1.Ordnung. Nach experimenteller Feinjustierung der Parameter empfehlen wir für Simulationen unter GPRS folgende Einstellungen:

Darstellungsmodell:	DIS-DR 1.Ordnung mit Time Compensation, begrenzt auf 600 ms
Threshold Position:	10% der Größe des Spielobjekts (1 Pixel)
Threshold Orientierung:	1°
Konvergenzmodell:	Lineare Konvergenz mit jeweils 250 ms Konvergenzzeit

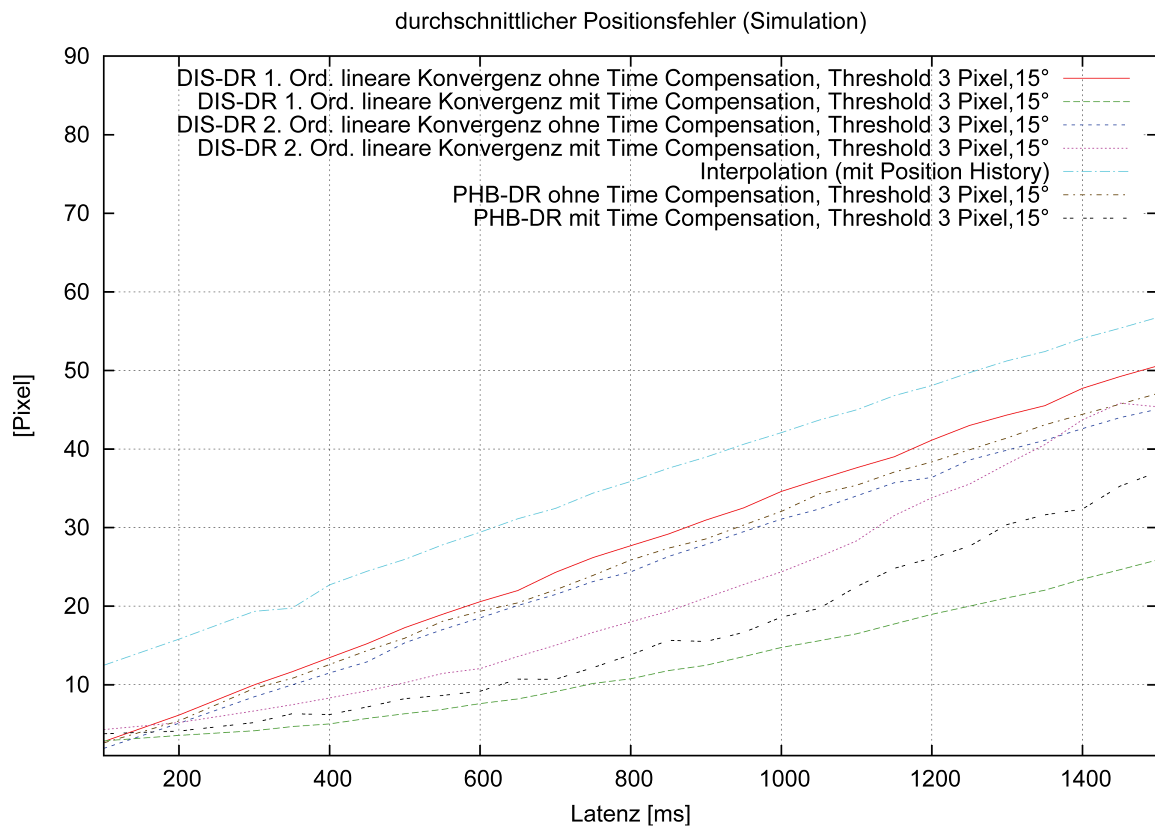


Abbildung 3.24: durchschnittliche Abweichung zur tatsächlichen Position (Simulation)

Mit diesen Einstellungen wurden folgende Werte erzielt:

Bewegungsverlauf:	gut - befriedigend (2.5)
durchschnittl. Senderate (Nutzdaten):	0,41 kByte/s
maximale Senderate (Nutzdaten):	0,734 kByte/s
durchschnittl. Positionsfehler:	15,56 Pixel (155 % der Größe des Spielobjekts)
maximaler Positionsfehler: 21,04 Pixel	(210 % der Größe des Spielobjekts)

Der Positionsfehler ist damit unter GPRS zu hoch, um für Rennspiele und Simulationen eine vernünftige Kollisionsabfrage realisieren zu können. Die *Trackmania*-Serie hat gezeigt, dass Rennspiele auch trotz dieser Einschränkung erfolgreich sein können. Jedoch muss die mögliche Spieleranzahl unter GPRS auf max. acht Spieler eingeschränkt werden, um die Datentransferrate innerhalb der verfügbaren Bandbreite zu halten (vgl. Abschnitt 2.4.1).

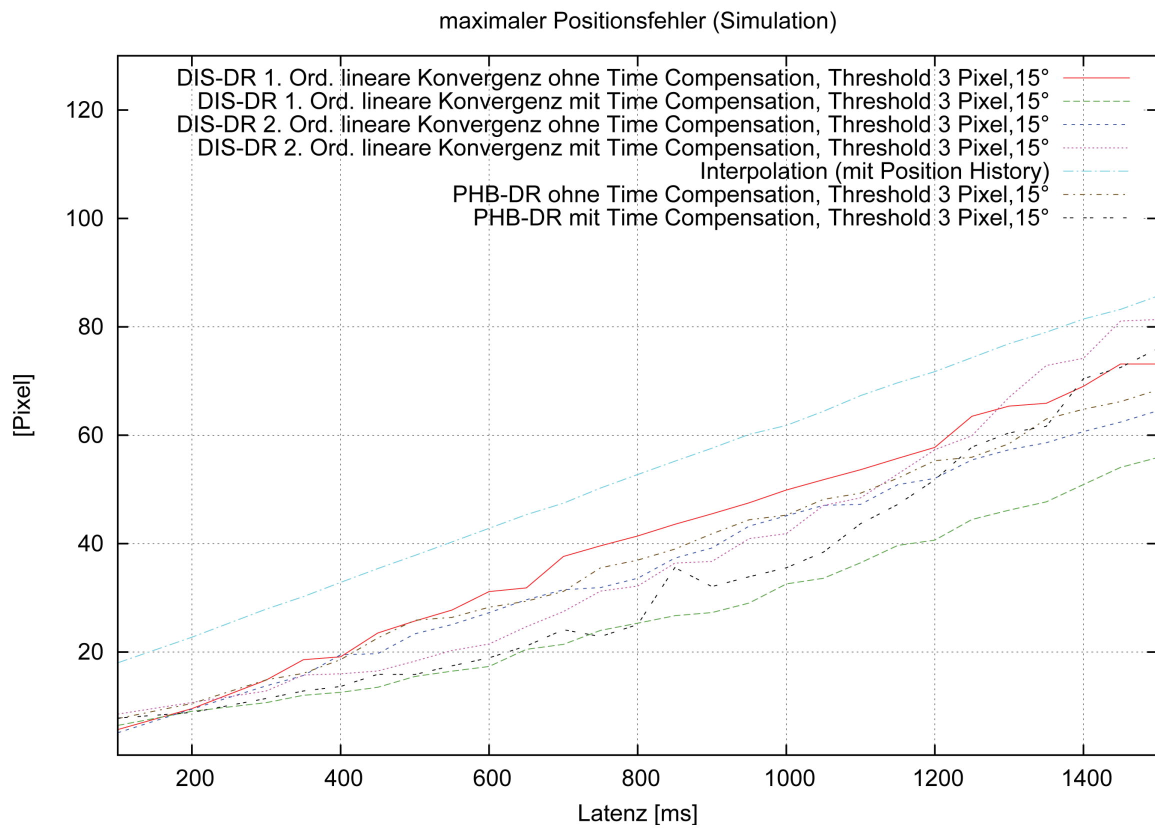


Abbildung 3.25: maximale Abweichung zur tatsächlichen Position (Simulation)

Simulationen unter UMTS

Verteilte Simulationen und Rennspiele sind auf Handys mit UMTS gut möglich. Durch die niedrigen Latenzen wird ein sauberer Bewegungsverlauf auch unter voller Time Compensation erzielt. Das führt unter DIS-DR zu einem sehr niedrigen durchschnittlichen Positionsfehler, wobei DIS-DR 1. und 2. Ordnung hier nahezu gleichauf liegen. In unserem Preset schnitt DR 1. Ordnung im Hinblick auf den Bewegungsverlauf und den durchschnittlichen Positionsfehler marginal besser ab. Nach experimenteller Feinjustierung der Parameter empfehlen wir für Simulationen unter UMTS folgende Einstellungen:

Darstellungsmodell:	DIS-DR 1.Ordnung mit Time Compensation, begrenzt auf 400 ms
Threshold Position:	10% der Größe des Spielobjekts (1 Pixel)
Threshold Orientierung:	1°
Konvergenzmodell:	Lineare Konv. mit Konv.zeit: 300 ms Position, 100 ms Orientierung

Mit diesen Einstellungen wurden folgende Werte erzielt:

Bewegungsverlauf:	gut (2.0)
durchschnittl. Senderate (Nutzdaten):	0,426 kByte/s
maximale Senderate (Nutzdaten):	0,642 kByte/s
durchschnittl. Positionsfehler:	5,01 Pixel (50 % der Größe des Spielobjekts)
maximaler Positionsfehler: 10,14 Pixel	(101 % der Größe des Spielobjekts)

Die niedrigen Positionsfehler stehen für eine hohe Konsistenz und damit für die Möglichkeit, Simulationsspiele mit Kollisionen und Projektilwaffen zu realisieren.

3.3.3 3D-Shooter

Das 3D-Action Steuerungsmodell repräsentiert 3D-Shooter wie Counter-Strike und Quake. Da die Testapplikation jedoch keine 3D sondern 2D-Grafik unterstützt, ist eine gewisse Abstraktion nötig: So wird die Entität statt aus der Ego-Perspektive aus der Vogel-Perspektive betrachet. Weiterhin fehlt die Maus als Eingabegerät, mit der es auf dem PC möglich ist, bequem die Orientierung zu ändern. Durch die Vereinfachung auf ein 2D Modell kann die Rotation der Orientierung jedoch auch mittels Tastendruck geschehen. Die Winkelgeschwindigkeit der Rotation ist dabei doppelt so hoch (180°/s) wie beim Simulations-Preset. Eine höhere Winkelgeschwindigkeit oder gar ein direktes Setzen der Orientierung wie beim Sportspiel-Preset macht keinen Sinn, da das Zielen auf gegnerische Entitäten sehr viel schwieriger wird. Weiterhin unterliegen beim 3D-Shooter-Preset die Entitäten einer sehr geringen Trägheit, und können dadurch noch extremere Richtungswechsel vollbringen als beim Sportspiel. Für den Test wurde die Bewegung der Entität dem Verhalten in einer Gefechtssituation entsprechend ausgeführt. Es wurden viele seitliche Ausweichschritte und schnelle Orientierungsänderungen getätigt, um den Gegner ins Visier zu nehmen. Wie unter 3.3 erläutert, ist es bei 3D-Shootern besonders wichtig, dass der Positionsfehler möglichst gering ausfällt, um ein Abschießen der Gegner durch Projektil-Waffen zu ermöglichen.

Bewegungsverlauf

Da das 3D-Shooter-Preset noch abruptere Richtungsänderungen als das Sportspiel-Preset erlaubt, wurde erwartet, dass nahezu alle Techniken in Zusammenarbeit mit unbegrenzter Time Compensation völlig untauglich sind. Diese Annahme bestätigt sich anhand der Messdaten (siehe Abb. 3.26). Schon bei 100 ms Latenz lässt sich die bei „DIS-DR zweiter Ordnung“ mit linearer oder Snap-Konvergenz, oder „PHB-DR“ entstehende Bewegung der entfernten Entität nur noch als mangelhaft beurteilen. Auch „DIS-DR erster Ordnung mit Snap-Konvergenz“ ist schon bei 100ms Latenz nahezu mangelhaft. „DIS-DR erster Ordnung

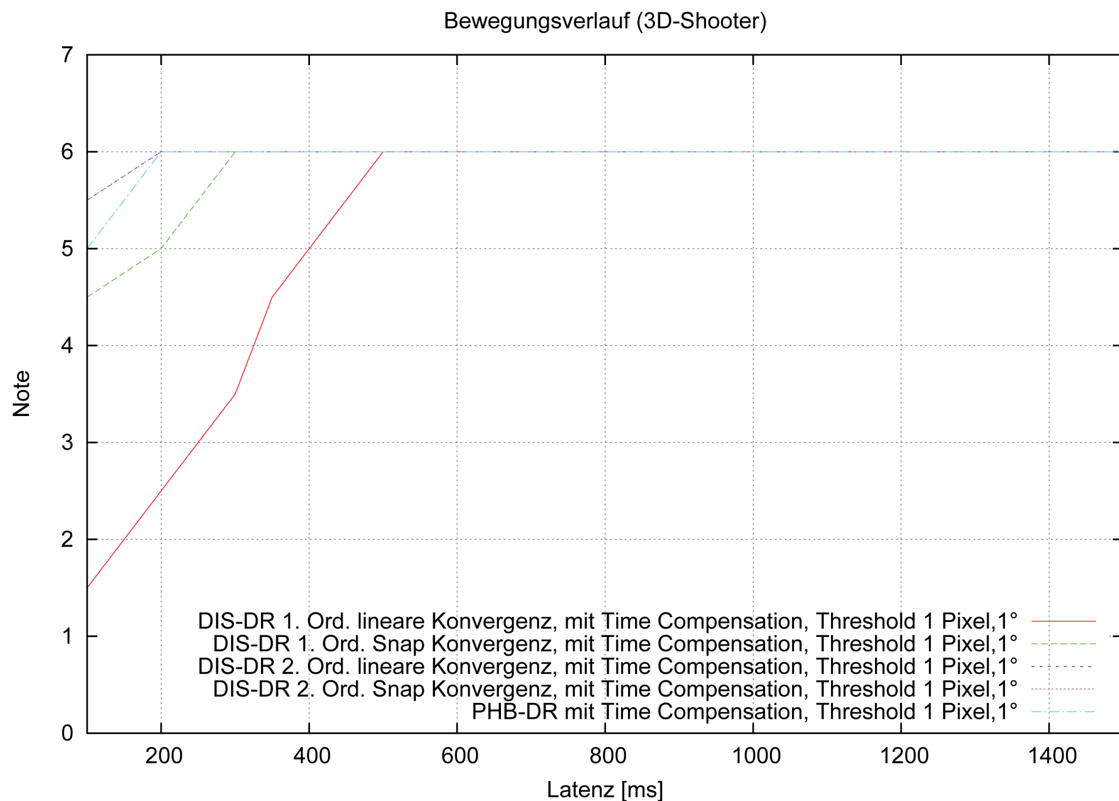


Abbildung 3.26: Authentizität des Bewegungsverlaufs bei 3D-Shootern (Techniken mit Time Compensation)

mit linearer Konvergenz“ ist die einzige Technik, die sich in Zusammenarbeit mit Time Compensation in Bezug auf die Bewegungsauthentizität bei 3D-Shootern als brauchbar erweist. Sie erzeugt bei einer Latenz von 100 ms „sehr gute“ und bei 200 ms „gute“ Werte. Bei 300 ms Latenz ist die Bewegung noch befriedigend und wird erst ab 400ms mangelhaft, was darauf hoffen lässt, dass „DIS-DR erster Ordnung mit Time Compensation“ für 3D-Shooter unter UMTS vielleicht doch genutzt werden kann.

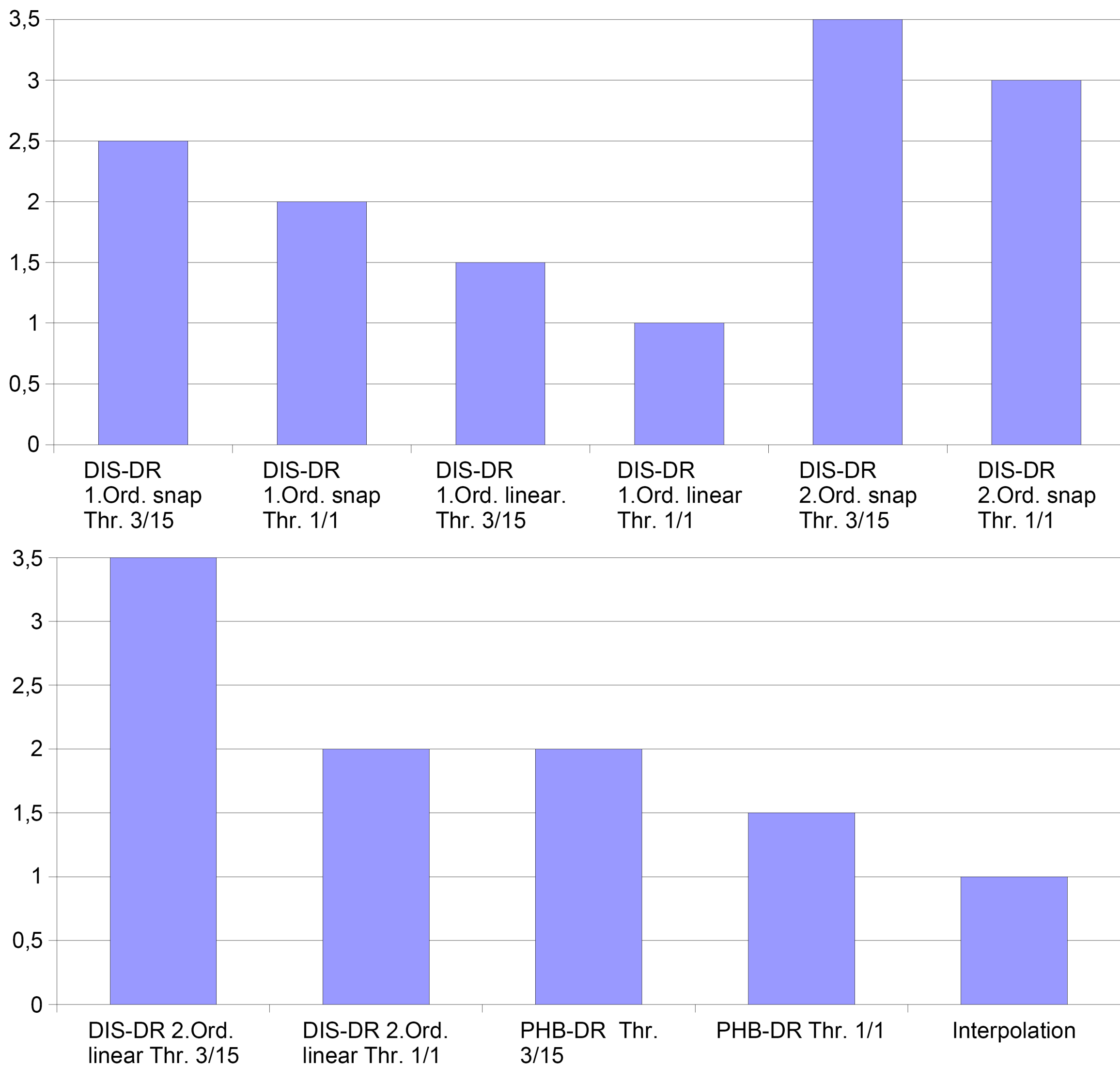


Abbildung 3.27: Authentizität des Bewegungsverlaufs bei 3D-Shootern, Techniken ohne Time Compensation

Wie bei der Auswertung des Sportspiels verändert sich die Authentizität der Bewegung bei steigender Latenz und Verwendung von Darstellungsmodellen ohne Time Compensation auch bei 3D-Shootern nicht. Die Bewertungen für jedes Darstellungsmodell lassen sich so

in einem Diagramm direkt gegenüberstellen (Abb. 3.27). Eine perfekte Nachahmung der Bewegung der lokalen Entität liefern hierbei die Techniken „DIS-DR erster Ordnung, lineare Konvergenz und Threshold 1/1“ und „Interpolation“ die mit einer 1 bewertet werden. Die nächst bessere Bewertung, mit einer Note von 1,5, wird von „DIS-DR erste Ordnung, lineare Konvergenz und Threshold 3/15“ und „PHB-DR mit Threshold 1/1“ erreicht. Der Bewegungsverlauf von „PHB-DR ist“ dabei etwas unruhiger als der von „DIS-DR erster Ordnung“, aber immer noch sehr gut. Ein gutes Bewegungsverhalten weisen die Techniken „DIS-DR 1st. Ord. Snap mit Threshold 1/1“, „DIS-DR 2nd. Ord. Linear mit Threshold 1/1“ und „PHB-DR mit Threshold 3/15“ auf. Wobei bei „DIS-DR 1st. Ord. Snap“ ein leichtes konstantes Vibrieren der Entität wahrzunehmen ist. Nahezu alle Techniken mit Snap-Konvergenz und die Technik „DIS-DR 2nd. Ord. Linear mit Threshold 3/15“ liefern den schlechtesten Bewegungsverlauf. Ein größerer Threshold sorgt im allgemeinen dafür, dass sich die Bewertung des Bewegungsverlaufes der meisten Techniken um einen halben Notenpunkt verschlechtert. Dead-Reckoning-Modelle mit Konvergenzalgorithmus und hoher Updaterate erzeugen also neben „Interpolation“ das beste Bewegungsverhalten.

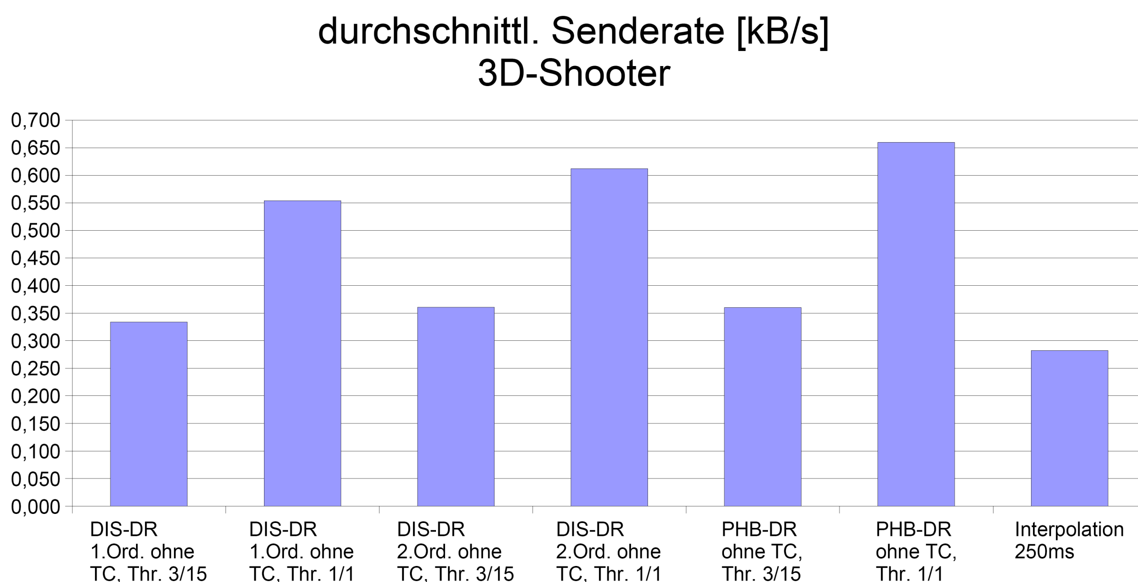


Abbildung 3.28: durchschnittliche Senderate ohne Time Compensation (3D-Shooter)

Datenaufkommen

Anhand der Abbildung 3.28 lässt sich erkennen, dass „Interpolation“ das im Bezug auf das Datenaufkommen günstigste Darstellungsmodell ist. Durchschnittlich werden bei „Interpolation“ 0,282 kB und maximal 0,32 kB an Daten pro Sekunde versendet. Die drei folgenden Expositiosmodelle „DIS-DR 1.Ord. Thr. 3/15“, „DIS-DR 2.Ord. Thr. 3/15“ und „PHB-DR Thr.

3/15“ unterscheiden sich bei der durchschnittlichen Senderate nur minimal voneinander, und liefern alle einen Wert um 0,35 kB/s. Die maximale Senderate unterscheidet sich ebenfalls sehr wenig, mit jeweils folgenden Werten: 0,486, 0,473 und 0,460 kB/s (siehe Abb. 3.30). Weiter ist zu erkennen, dass durch die Verwendung eines sehr kleinen Thresholds von 1/1, die durchschnittlichen wie maximalen Senderaten der Darstellungsmodelle um ca. 65% bei „DIS-DR 1.Ord.“, um ca. 70% bei „DIS-DR 2.Ord.“ und um ca. 85% bei „PHB-DR“ ansteigen. Auch die Verwendung von Time Compensation sorgt für ein höheres Datenaufkommen, da zusätzlich ein Zeitstempel übertragen werden muss (siehe Abschnitt 3.3). Abbildung 3.29 zeigt, dass sich das Datenaufkommen um ca. 20% bei „DIS-DR 1.Ord.“ und um ca. 10% bei „DIS-DR 2.Ord.“ erhöht hat. Da „PHB-DR“ eine Übertragung des Zeitstempels schon voraussetzt, hat die Verwendung von Time Compensation keinen Einfluss auf das Datenaufkommen. Abschließend lässt sich zusammenfassen, dass „Interpolation“ das Darstellungsmodell mit dem geringsten Datenaufkommen ist, der Unterschied zu den nächst besseren Modellen jedoch sehr gering ist. Da Darstellungsmodelle der Dead-Reckoning-Familie untereinander nahezu das gleiche Datenaufkommen aufweisen, lässt sich bei einer Wahl zwischen diesen Modellen die Senderate als Kriterium vernachlässigen.

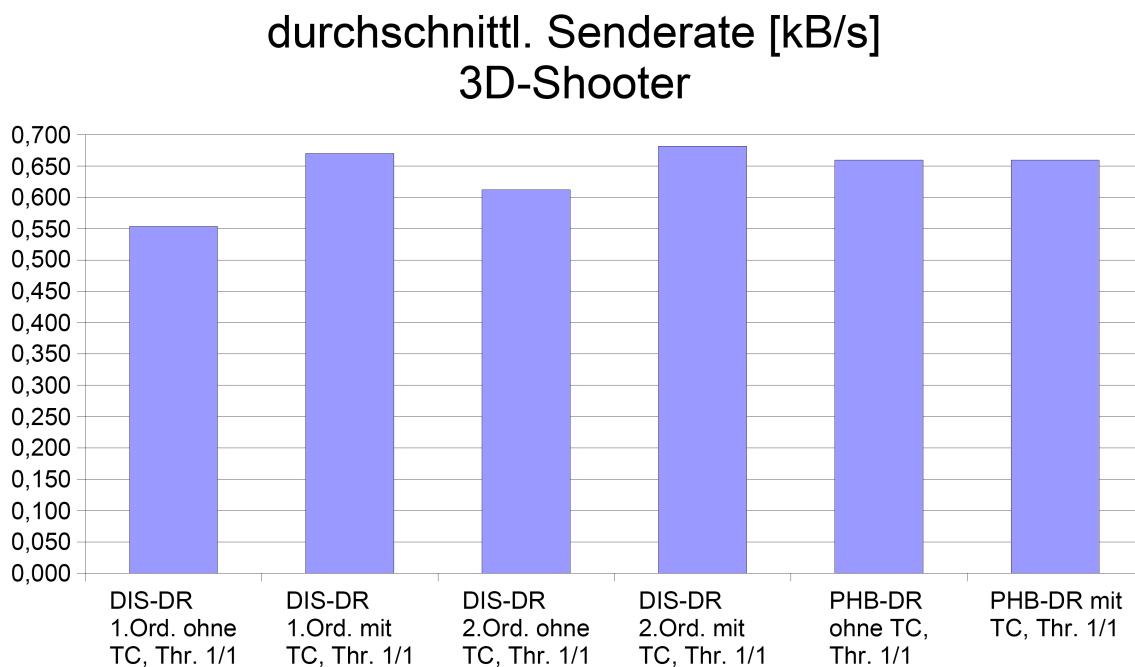


Abbildung 3.29: durchschnittliche Senderate mit Time Compensation (3D-Shooter)

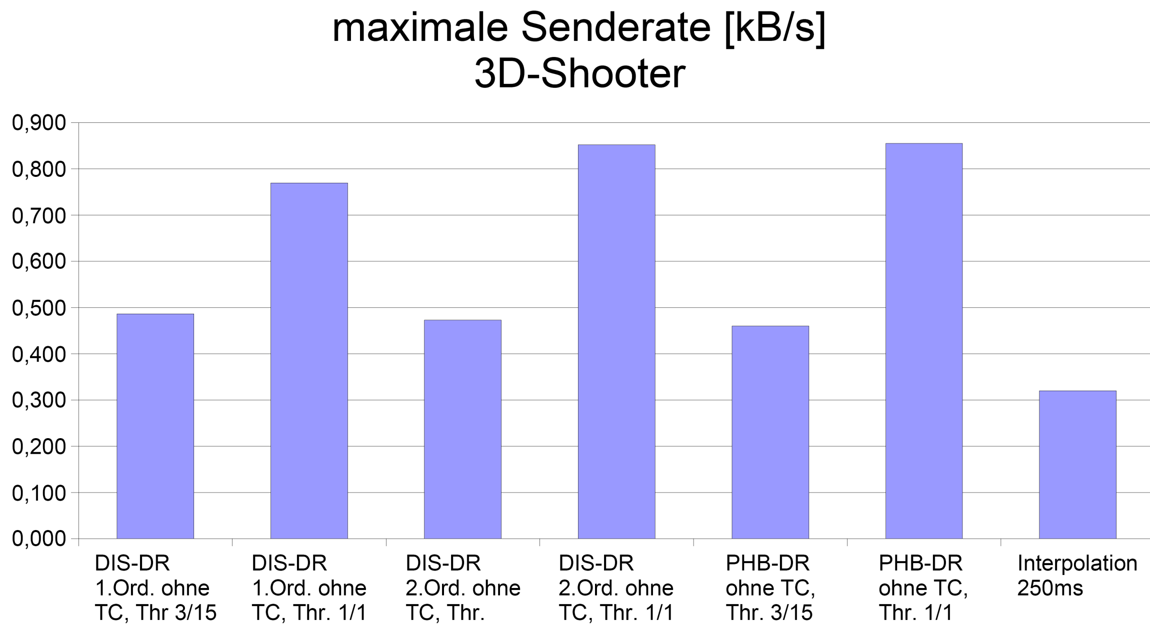


Abbildung 3.30: maximale Senderate (3D-Shooter)

Positionsfehler

Um die verschiedenen Darstellungsmodellen aufgrund des bei 3D-Shootern entstehenden Positionsfehler zu bewerten, ist es zuerst notwendig, einen internen Vergleich zwischen den Dead-Reckoning-Darstellungsmodellen durchzuführen.

In Abb. 3.31 wird dargestellt, wie sich der Positionsfehler für „DIS-DR erster Ordnung“ mit Snap- oder linearer Konvergenz bei einer Umstellung des Thresholds von 3/15 auf 1/1 verbessert. Bis zu einer Latenz von 600 ms ist die Verbesserung des Positionsfehlers bei beiden Konvergenzmodellen gleich, und beträgt ungefähr durchschnittlich einen Pixel. Die durchschnittliche Gesamtverbesserung beträgt bei Snap-Konvergenz 1,4 Pixel und bei linearer Konvergenz 0,7 Pixel. Eine Einstellung des Thresholds von 1/1 statt 3/15 wirkt sich also für „DIS-DR erster Ordnung“ insgesamt stärker bei Snap, als bei linearer Konvergenz aus.
In Abb. 3.32 wird die Verbesserung des Positionsfehlers für „DIS-DR zweiter Ordnung“ bei beiden Konvergenzmodellen dargestellt. Bei Snap-Konvergenz verbessert sich der Positionsfehler über den gesamten Verlauf nur minimal um durchschnittlich 0,26 Pixel. Bei linearer Konvergenz verbessert sich der Positionsfehler bis 600 ms auch nur sehr gering um durchschnittlich 0,28 Pixel, ab einer Latenz von 650 ms jedoch um wesentlich mehr, was dazu führt, dass die durchschnittliche Gesamtverbesserung 1,86 Pixel beträgt. Bei „DIS-DR zwei-

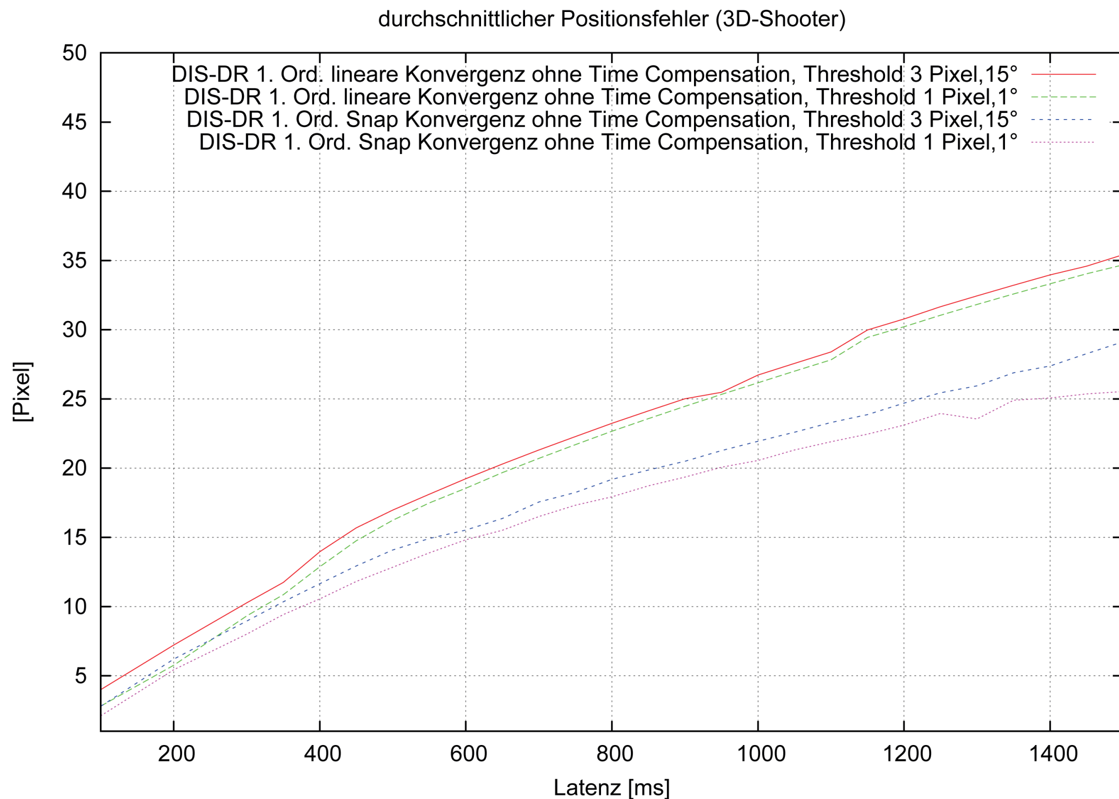


Abbildung 3.31: Threshold Vergleich: DIS-DR erste Ordnung

ter Ordnung“ wirkt sich eine Einstellung des Thresholds auf 1/1, statt 3/15, also insgesamt stärker auf lineare Konvergenz als auf Snap-Konvergenz aus.
Für „PHB-DR“ hat sich eine Umstellung des Thresholds von 3/15 auf 1/1 als unnötig herausgestellt, da die Verbesserung des Positionsfehlers verschwindend gering ist.
Zusammenfassend lässt sich sagen, dass bei 3D-Shootern eine Umstellung des Thresholds eher für „DIS-DR erster Ordnung“ als für „DIS-DR zweiter Ordnung“ Sinn macht.

In Abb. 3.33 sind alle „DIS-DR“-Verfahren mit den verschiedenen Konvergenzmodellen und einem Threshold von 1/1 gegenübergestellt. Es lässt sich deutlich erkennen, dass „DIS-DR erster Ordnung mit linearer Konvergenz“ am schlechtesten abschneidet. Die anderen drei Kombinationen der Verfahren, liegen bis zu einer Latenz von 400 ms nahezu gleich auf. „DIS-DR 2. Ord. Snap“ verschlechtert sich ab dieser Latenzzeit im Vergleich zu „DIS-DR 2. Ord. linear“ und „DIS-DR 1. Ord. Snap“. Bei 450 ms Latenz verschlechtert sich „DIS-DR 2. Ord. linear“ abrupt und liegt bis zu einer Latenz von 650 ms gleichauf mit „DIS-DR 2. Ord. Snap“. Ab einer Latenz von 450 ms ist „DIS-DR 1. Ord. Snap“ also das Verfahren, das den geringsten Positionsfehler erzeugt. Der durchschittliche Positionsfehler, den die besseren

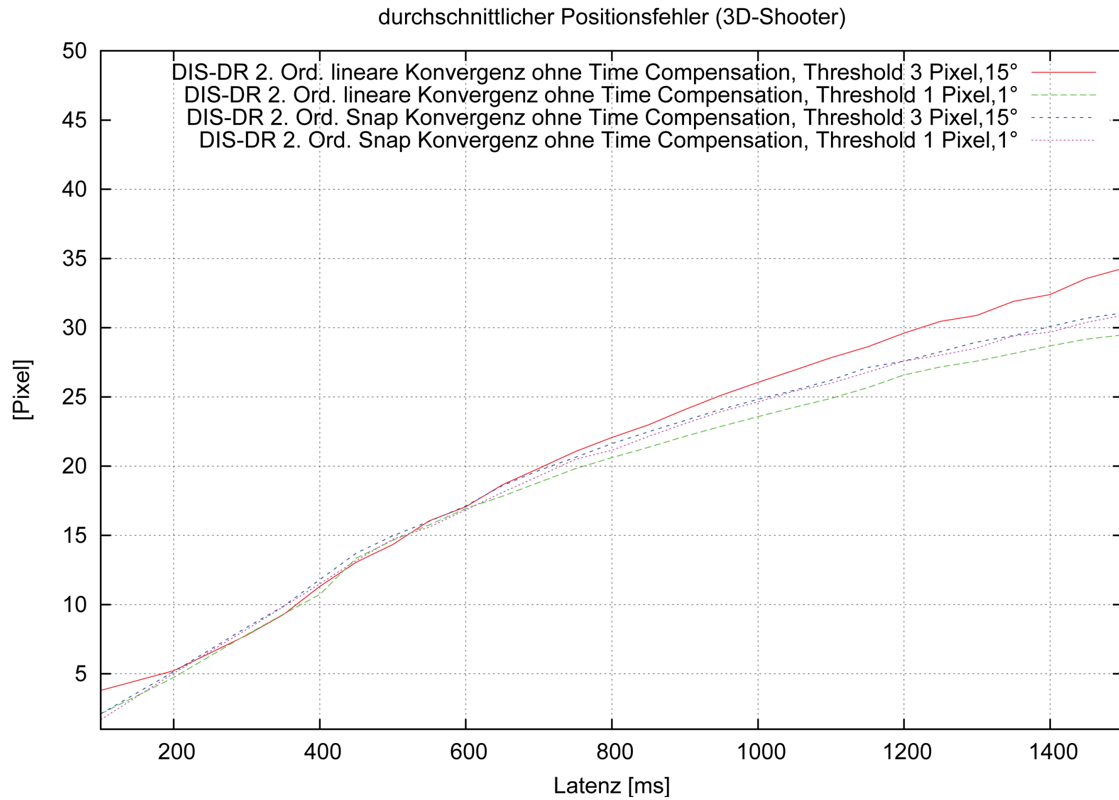


Abbildung 3.32: Threshold Vergleich: DIS-DR zweite Ordnung

drei Verfahren bis zu einer Latenz von 400 ms erzeugen, beträgt 7,1 Pixel bei „DIS-DR 1. Ord. Snap“, 6,94 Pixel bei „DIS-DR 2. Ord. linear“ und 7,25 Pixel bei „DIS-DR 2. Ord. Snap“. Falls also Latenzzeiten bis maximal 400 ms anfallen, ist „DIS-DR 2. Ord. linear“ das beste Verfahren im Bezug auf den Positionsfehler. Fallen jedoch höhere Latenzzeiten an, ist „DIS-DR 1. Ord. Snap“ das bessere Verfahren.

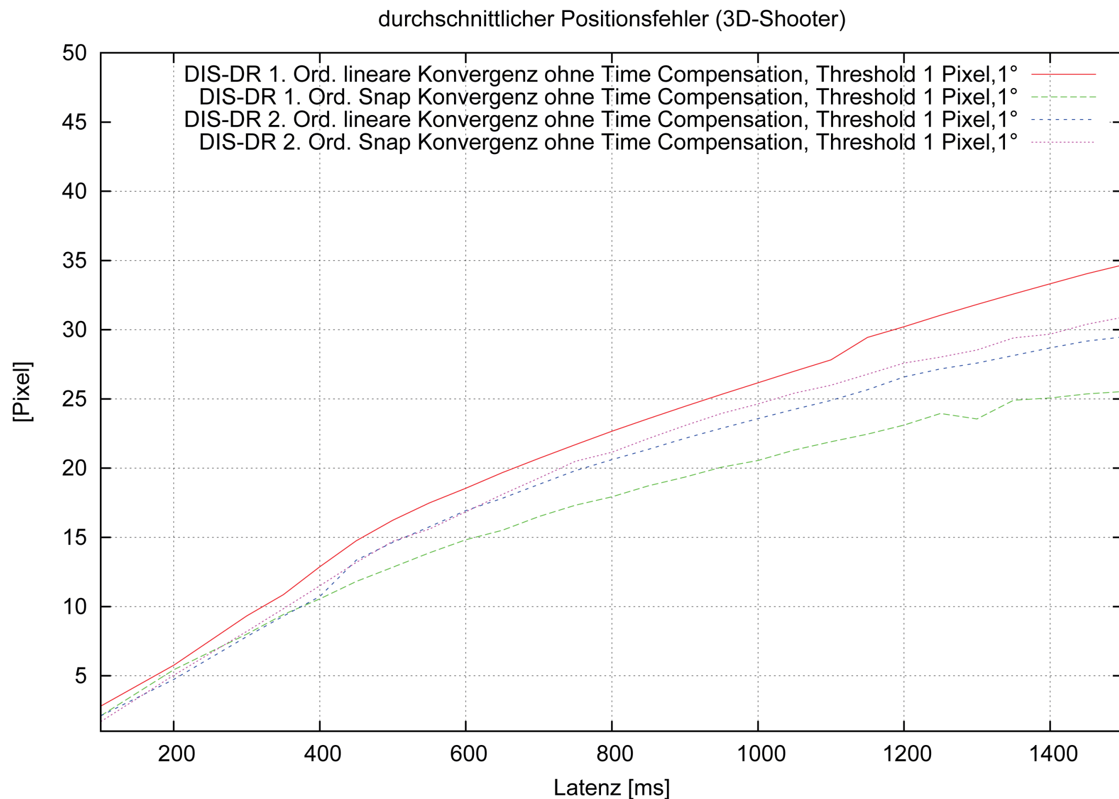


Abbildung 3.33: DIS-DR Threshold 1/1 Vergleich

Aufgrund der vorherigen Erkenntnisse wurden folgende Darstellungsmodelle für eine direkte Gegenüberstellung ausgewählt:

- DIS-DR erster Ordnung, Snap-Konvergenz, mit und ohne Time Compensation, Threshold 1/1
- DIS-DR zweiter Ordnung, lineare Konvergenz, mit und ohne Time Compensation, Threshold 1/1
- PHB-DR, mit und ohne Time Compensation, Threshold 3/15
- Interpolation

Es ist deutlich zu erkennen, dass Darstellungsmodelle wie „DIS-DR zweiter Ordnung mit linearer Konvergenz“ und „PHB-DR“, die die Beschleunigung der Entität berücksichtigen und zusätzlich Time Compensation einsetzen, schon bei geringer Latenz wesentlich schlechter abschneiden als die anderen. Auch „Interpolation“ ist in Bezug auf den durchschnittlichen

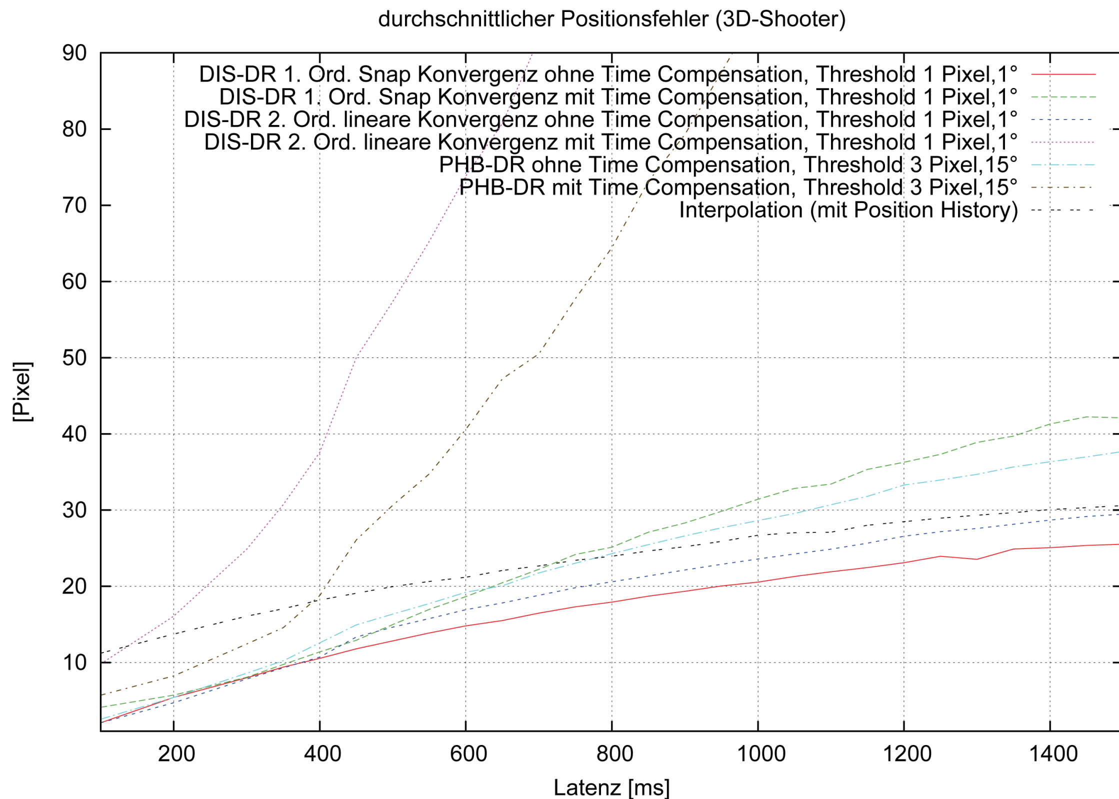


Abbildung 3.34: durchschnittliche Abweichung zur tatsächlichen Position (3D-Shooter)

Positionsfehler bis zu einer Latenz von 750 ms sehr viel schlechter als die anderen Darstellungsmodelle. Der durchschnittliche Positionsfehler aller weiteren Darstellungsmodelle unterscheidet sich allerdings bis zu einer Latenz von 400 ms kaum.
Ab einer Latenz von 450 ms erzeugt „DIS-DR erster Ordnung mit Snap-Konvergenz ohne Time Compensation“ den kleinsten durchschnittlichen Positionsfehler, mit einigem Abstand folgt „DIS-DR zweiter Ordnung, linearer Konvergenz ohne Time Compensation“. Zusammenfassend lässt sich sagen, dass bis zu einer Latenz von 400 ms „DIS-DR zweiter Ordnung, lineare Konvergenz, ohne Time Compensation und Threshold 1/1“, sowie „DIS-DR erster Ordnung, Snap-Konvergenz, ohne Time Compensation und Threshold 1/1“ die Techniken sind, die die kleinste Abweichung verursachen. Der durchschnittliche Positionsfehler beträgt respektive 6,94 und 7,1 Pixel, beide Modelle sind also gleichermaßen zu empfehlen. Auch bei der Betrachtung des maximalen Positionsfehler (Abb. 3.35) ist zu erkennen, dass diese beiden Modell über den gesamten Verlauf am besten abschneiden.

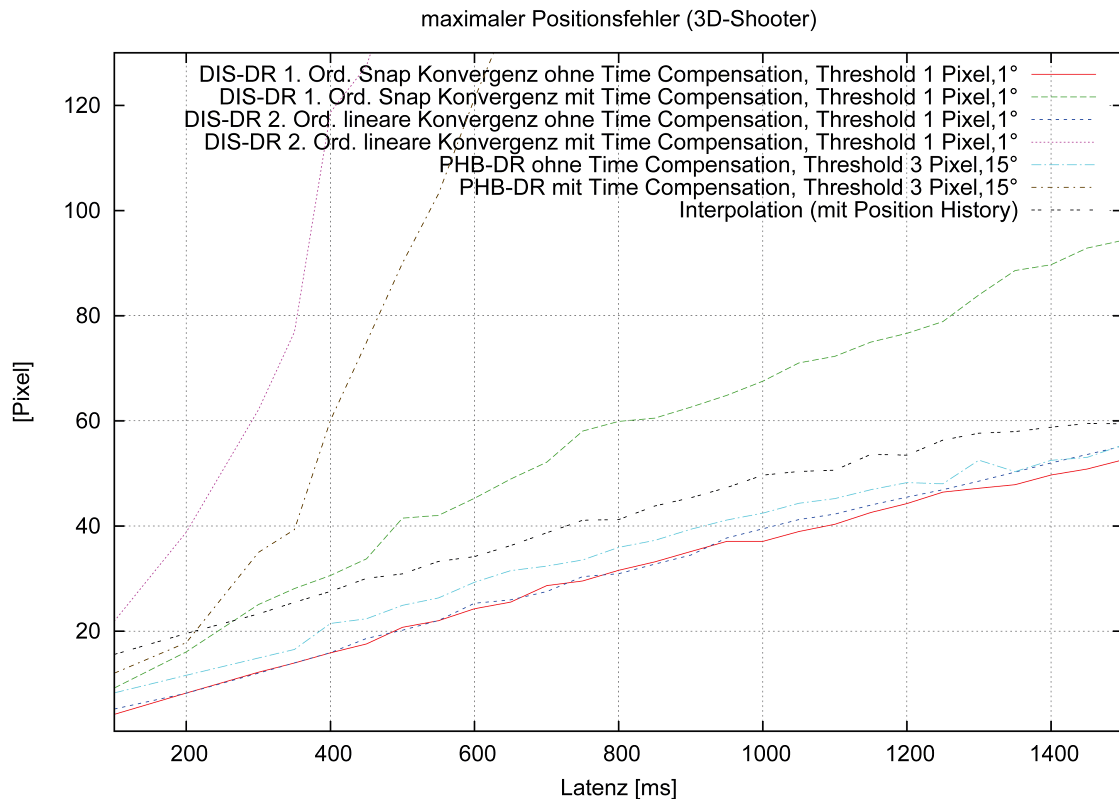


Abbildung 3.35: maximale Abweichung zur tatsächlichen Position (3D-Shooter)

Fazit 3D-Shooter

Die vorhergehenden Test ergaben, dass sich in Abhängigkeit zur Latenz folgende Darstellungsmodelle am besten für 3D-Shooter eignen:

- bis zu 400 ms Latenz: „DIS-DR zweiter Ordnung, lineare Konvergenz, ohne Time Compensation, Threshold 1/1“. Diese Technik liefert die größte Positionsübereinstimmung und weist eine gute Bewegungsbewertung auf. Es muss jedoch noch weiter untersucht werden, ob Darstellungsmodelle mit einer limitierten Time Compensation vielleicht besser abschneiden.

- über 450 ms Latenz: „DIS-DR erster Ordnung, Snap-Konvergenz, ohne Time Compensation, Threshold 1/1“. Dies ist die Technik, die unter hoher Latenz die mit Abstand kleinsten Positionsfehler erzeugt. Es ist jedoch sehr unwahrscheinlich, dass der Positionsfehler klein genug ist, um 3D-Shooter zufriedenstellend zu realisieren.

3D-Shooter unter GPRS

Genau wie Sportspiele sind 3D-Shooter unter GPRS nicht zu realisieren. Der kleinste durchschnittliche Positionsfehler, der mittels „DIS-DR erster Ordnung, Snap-Konvergenz und Threshold 1/1“ erzeugt wird, beträgt bei 800 ms Latenz 17,93 Pixel. Beim Schießen mit Projektilwaffen auf eine entfernte Entität, muss also durchschnittlich um eine Distanz vorgehalten werden, die 180% größer ist als die eigentliche Entität. Der bei 800 ms Latenz aufgetretene maximale Positionsfehler beträgt 31,59 Pixel, es kann also sein, dass sich die zu vorhaltende Distanz auch auf das 3-fache der Entitätsgröße beläuft. Diese Werte belegen, dass die Trefferwahrscheinlichkeit mit Projektilwaffen sehr schlecht ist. Erschwerend kommt hinzu, dass bei 3D-Shootern der Abschuss von Projektilwaffen vom Server bestätigt werden muss (siehe Abschnitt 2.8.3). Erst bei Erhalt der Bestätigung wird das Projektil lokal visualisiert. Die Zeitdauer vom Tastendruck bis zum eigentlichen Abschuss einer Projektilwaffe liegt bei GPRS mit 800 ms Latenz also weit über dem als unmerklich ermittelten Wert (siehe Abschnitt 2.5.1).

3D-Shooter unter UMTS

Um die für 3D-Shooter unter UMTS geeignetste Technik zu ermitteln, wurde eine Auswertung von 12 Testläufen mit den unter 2.4.5 angegebenen Netzwerkeinstellungen folgender Darstellungsmodelle durchgeführt:

- 1: DIS-DR zweiter Ordnung, lineare Konvergenz, ohne Time Compensation, Threshold 1/1
- 2: DIS-DR zweiter Ordnung, lineare Konvergenz, mit Time Compensation von maximal 100 ms, Threshold 1/1
- 3: DIS-DR erster Ordnung, lineare Konvergenz, mit Time Compensation von maximal 100 ms, Threshold 1/1.

Darstellungsmodell:	1	2	3
Bewegungsverlauf:	gut (2)	ausreichend (4)	sehr gut (1,5)
durchschnittl. Senderate:	0,534 kByte/s	0,595 kByte/s	0,553 kByte/s
maximale Senderate:	0,757 kByte/s	0,878 kByte/s	0,841 kByte/s
durchschnittl. Positionsfehler:	10,65 Pixel	7,3 Pixel	9,7 Pixel
maximaler Positionsfehler:	20,88 Pixel	21,30 Pixel	19,65 Pixel

Obwohl „DIS-DR zweiter Ordnung, lineare Konvergenz, mit Time Compensation von maximal 100 ms, Threshold 1/1“ den kleinsten durchschnittlichen Positionsfehler liefert, ist dieses Modell aufgrund des nur als „Ausreichend“ bewerteten Bewegungsverlaufes ungeeignet für 3D-Shooter. Im Bezug auf den durchschnittlichen und maximalen Positionsfehler, liefert

„DIS-DR erster Ordnung, lineare Konvergenz, mit Time Compensation von maximal 100 ms, Threshold 1/1“ auch nur etwas bessere Daten als „DIS-DR zweiter Ordnung, lineare Konvergenz, ohne Time Compensation, Threshold 1/1“. Da aber auch der Bewegungsverlauf besser bewertet wurde, ist dies die Technik, die unter realen Netzwerkbedingungen einer UMTS-Verbindung am besten abschneidet.
Es wurden bei einer Latenz von 340 ms ein durchschnittlicher Positionsfehler von 9,7 Pixel und ein maximaler Positionsfehler von 19,65 Pixel ermittelt. Beim Schießen mit Projektilwaffen auf gegnerische Entitäten muss also um durchschnittlich das 0,97-fache, bzw. maximal das 2-fache der Entitätsgröße vorgehalten werden. Dies sind Werte, die das Spielgefühl schon relativ stark beeinträchtigen dürften. Die Verzögerung des Abfeuerns von Projektilwaffen um 340 ms ist zwar auch größer als der empfohlene Wert von 100 ms (siehe Abschnitt 2.5.1), dürfte sich in der Praxis aber unserer Meinung nach noch als hinnehmbar erweisen.
Im Bezug auf die Bandbreite gibt es zumindest unter UMTS keine Hindernisse: Der Downlink von 105 kbit/s (siehe Tabelle 2.2) wird selbst von 8 Spielern bei höchstem Datenaufkommen (8 * 0,85 kB/s * 8 = 54,4 kbit/s) nur zur Hälfte ausgereizt.
Ingesamt ist die Realisierbarkeit von 3D-Shootern unter UMTS also nur mit einigen Einschränkungen möglich. Durch die Verwendung von Lag Compensation (2.8.2) bzw. clientseitiger Trefferauswertung wäre der Positionsfehler unererheblich, aus Sicht der beschossenen Entitäten würden jedoch Paradoxien (siehe Abschnitt 2.8.2) entstehen. Des weiteren funktioniert Lag Compensation nur mit Instant-Hit-Waffen, ein 3D-Shooter wäre also sehr eingeschränkt in seinem Waffensortiment. Um dennoch Projektil-Waffen zu unterstützen, wäre die einzige Alternative die Entitäten träger zu machen, sodass der Positionsfehler kleiner wird. Weitere Tests sollten untersuchen, ab welcher Trägheit ein für Projektilwaffen akzeptabler Positionsfehler entsteht.

Zu empfehlendes Darstellungsmodell für 3D-Shooter unter UMTS:

Darstellungsmodell:	DIS-DR 1.Ordnung mit Time Compensation, begrenzt auf 100 ms
Threshold Position:	10% der Größe des Spielobjekts (1 Pixel)
Threshold Orientierung:	1°
Konvergenzmodell:	Lineare Konvergenz mit jeweils 125 ms Konvergenzzeit

3.3.4 Arcade-Action

Das Arcade-Action-Preset stellt Echtzeit-Multiplayer-Puzzlespiele wie *Bomberman* oder *Pac-Man* dar. Entitäten dieser Spiele weisen oft eine extrem hohe Bewegungsgeschwindigkeit ohne Trägheit auf, und können so unmittelbar ihre Bewegungsrichtung umkehren. Das Spielfeld besteht meistens aus labyrinthartigen Strukturen, die die Entitäten in ihren Bewegungsmöglichkeiten einschränken. Ein geschicktes Ausnutzen dieser Strukturen mit den zur Verfügung stehenden Bewegungsmöglichkeiten, stellt den Reiz dieser Spiele dar. Es ist daher absolut

unbefriedigend, wenn entfernte Entitäten aufgrund des Darstellungsmodells eine unmögliche Bewegung, z. B. diagonal durch Wände hindurch, vollführen. Die engen Gänge des Labyrinths sorgen auch dafür, dass Entitäten häufig unausweichlich aufeinander treffen und sich den Weg abschneiden. Der Positionsfehler zwischen lokaler und entfernter Entität darf also nur minimal sein, damit Kollisionen fair erkannt werden können.
Aufgrund dieser harten Anforderungen ist eine Realisierung von Arcade-Action-Spielen, selbst unter einer UMTS-Verbindung, sehr unwahrscheinlich. Für den Test wurde das typische Verhalten eines Bombermans nachgeahmt: Längere Bewegungsphasen in eine Richtung werden von schnellen Ausfallbewegungen unterbrochen, um Bomben in die Gänge zu platzieren.

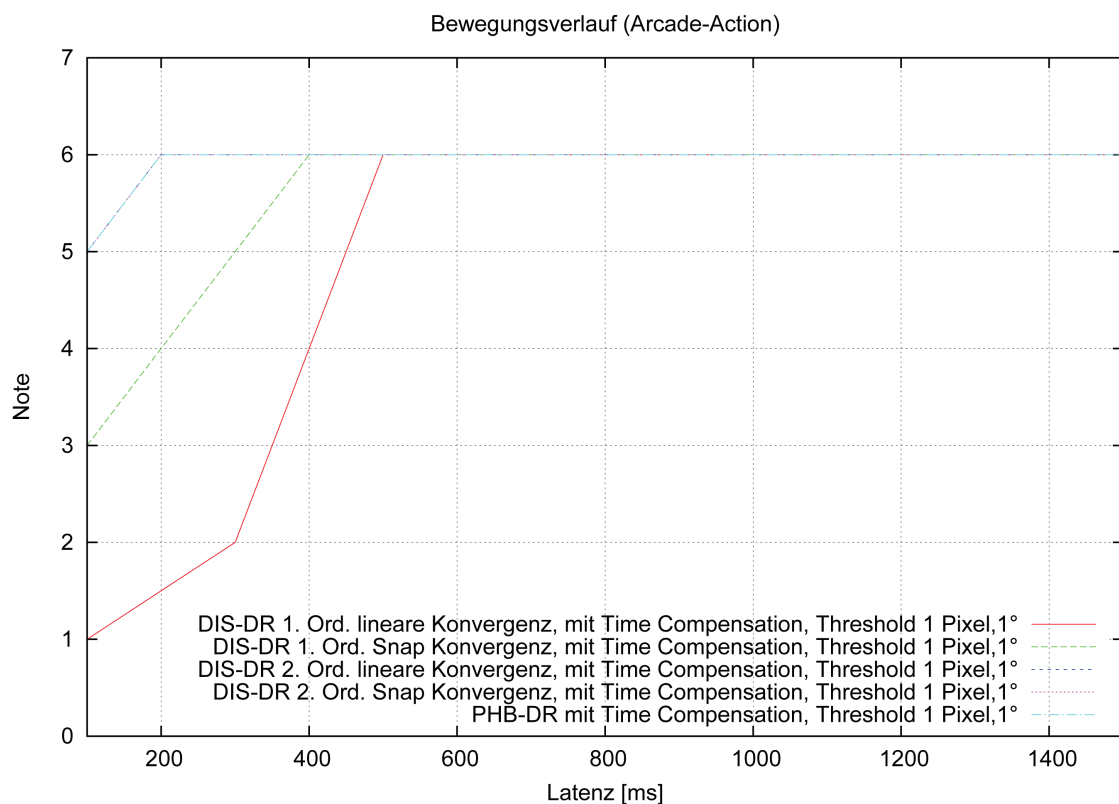


Abbildung 3.36: Authentizität des Bewegungsverlaufs bei Arcade-Action (Techniken mit Time Compensation)

Bewegungsverlauf

Das in Abbildung 3.36 bewertete Darstellungsmodell, das den angenehmsten Bewegungsverlauf bei aktivierter Time Compensation aufweist, ist „DIS-DR erster Ordnung mit linearer

Konvergenz". Bis zu einer Latenz von 200 ms ist die Bewegung noch sehr gut, und erst ab 350 ms erzeugt der Konvergenzalgorithmus merklich unrealistische diagonale Abweichungen. Die einzige Alternative ist „DIS-DR erster Ordnung mit Snap-Konvergenz", die zwar keine unrealistische diagonale Bewegung erzeugt, aber aufgrund der heftigen Sprünge der Entität ab 200 ms Latenz nur noch mit „ausreichend" bewertet werden kann. Die weiteren Techniken sind bei aktivierter Time Compensation schon ab 100 ms nur noch mangelhaft und daher nicht praxistauglich.

Auch bei Arcade-Action wirkt sich die Latenz bei deaktivierter Time Compensation nicht auf das Bewegungsverhalten der Darstellungsmodelle aus. Das Bewegungsverhalten der entfernten Entität ist über die gesamte Erhebung gleich, die Bewegung ist nur um die jeweilige Latenzzeit verzögert. Die Technik, die ohne Time Compensation am besten abschneidet, ist „DIS-DR 1. Ordung, lineare Konvergenz, Threshold 1/1" (siehe Abb. 3.37). Durch ein Setzen der Orientierungskonvergenzzeit auf 1 ms lässt sich auch der einzige Makel, das Rotieren der Entität, verhindern. Bis auf „DIS-DR 2. Ord., linear, Thr. 3/15" und „PHB-DR, Thr. 3/15" erzeugen die übrigen Modelle auch sehr gute bis gute Bewegungsverläufe. Alle Modelle mit linearer Konvergenz haben das Problem, dass sie mehr oder minder ausgeprägte unrealistische diagonale Bewegungen erzeugen. Dies ist bei Snap-Konvergenz nicht der Fall, dafür ist die Bewegung durch die vielen kleinen Sprünge, die als Vibration wahrgenommen werden, weitaus unruhiger. Bei „PHB-DR" ist es nicht möglich, die Orientierungskonvergenzzeit soweit zu minimieren, dass keine falschen Rotationen mehr enstehen. Da zusätzlich diagonal konvergiert wird, ist „PHB-DR" das Darstellungsmodell welches die unrealistischste Bewegung erzeugt.

Datenaufkommen

Das Darstellungsmodell welches am wenigsten Daten pro Sekunde generiert ist „DIS-DR, erster Ordnung, ohne TC, mit Threshold 3/15 " mit einer Datenrate von 0,26 kB/s. Damit erzeugt es um ca. 10% weniger Daten, als das nächst bessere Modell „Interpolation". Die prozentuale Erhöhung der Datenrate bei einer Umstellung des Threshold von 3/15 auf 1/1, beträgt bei „DIS-DR erster Ordnung" ca. 40%, und bei „DIS-DR zweiter Ordnung" und „PHB-DR" ungefähr 27% (siehe Abb. 3.38). Ein Aktivieren der Time Compensation erhöht die Datenrate bei „DIS-DR" erster und zweiter Ordnung respektive um weitere 7% bzw. 11%. Auf „PHB-DR" hat eine Aktivierung der Time Compensation keine Auswirkung.
Die maximale Datenrate hat sich, wie in Abb. 3.39 abgebildet, im Durchschnitt zwischen Threshold 3/15 und 1/1 bei „DIS-DR erster Ordnung" um das 2-fache, bei „DIS-DR zweiter Ordnung" um das 1,9-fache, bei „PHB-DR" um das 1,8-fache und bei „Interpolation" um das 1,25-fache erhöht. „Interpolation" ist also das Modell mit der beständigsten Datenrate. Bei den anderen Modellen kann zwischenzeitlich nahezu das Doppelte an Daten anfallen.

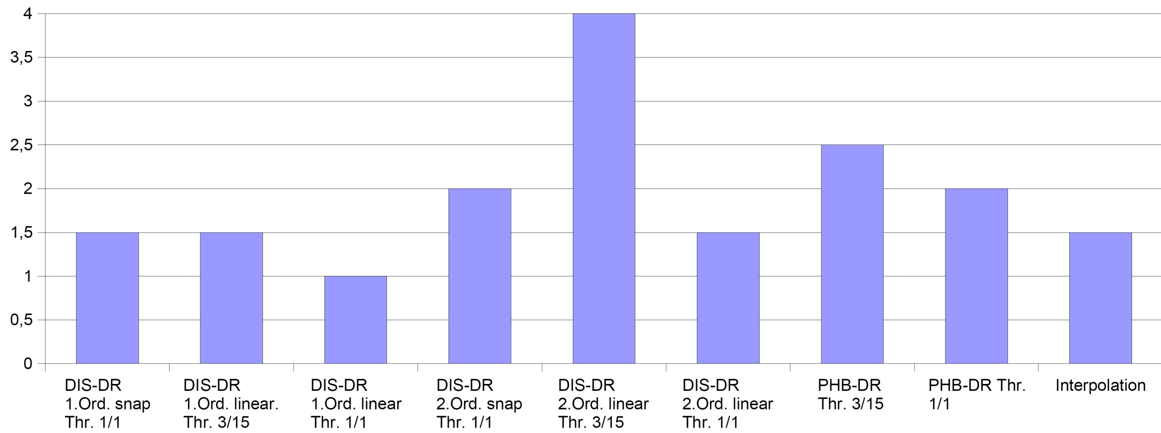


Abbildung 3.37: Authentizität des Bewegungsverlaufs bei Arcade-Action, Techniken ohne Time Compensation

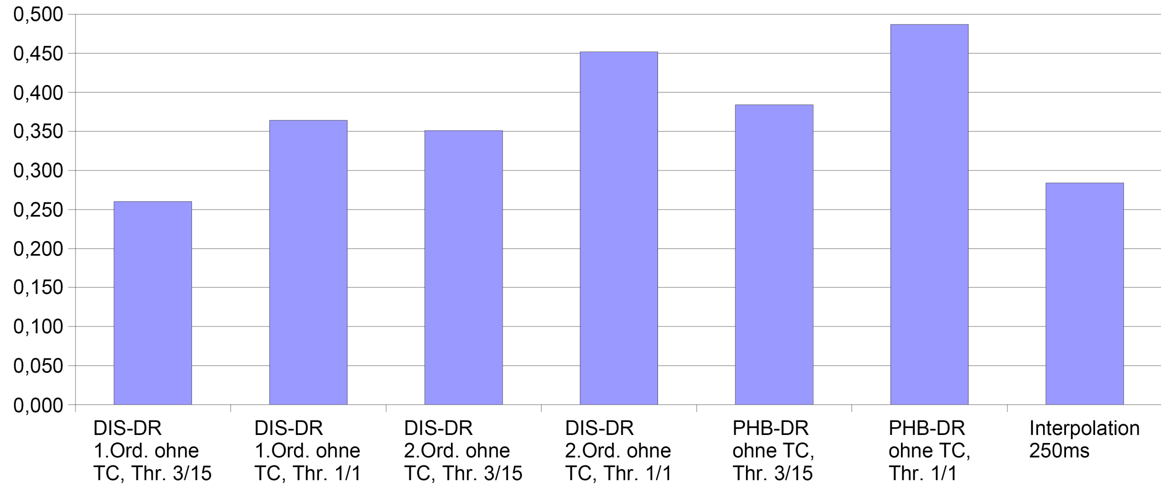


Abbildung 3.38: durchschnittliche Senderate (Arcade-Action)

Positionsfehler

Bei Arcade-Action-Spielen gibt es eine extrem hohe Anforderung an die Positionsgenauigkeit, und gleichzeitig eine, dieser Anforderung entgegenwirkende, hohe Bewegungsge-

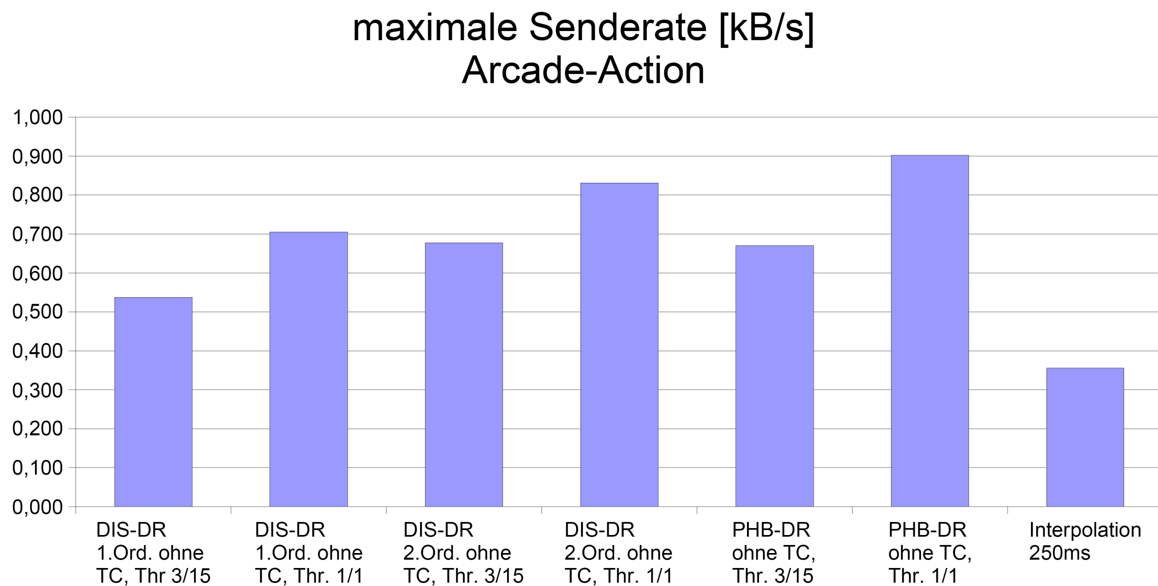


Abbildung 3.39: maximale Senderate (Arcade-Action)

schwindigkeit der Entitäten. Es werden daher für die Analyse der Darstellungsmodelle Thresholdeinstellungen von 3/15, die eine Verstärkung des Positionsfehlers verursachen, vernachlässigt. Außerdem können auch alle Darstellungsmodelle mit aktivierter Time Compensation bis auf „DIS-DR erster Ordnung“ vernachlässigt werden, da sie, wie unter Abb. 3.3.4 gezeigt, den Bewegungsverlauf sehr schlecht nachahmen.
In Abbildung 3.40 ist zu erkennen, dass sich fast alle Darstellungsmodelle im Hinblick auf den Positionsfehler bis zu einer Latenz von 200 ms kaum unterscheiden. „Interpolation“ ist das Modell, das schon zu Anfang wesentlich schlechter ist als alle anderen, und dies auch bis zu einer Latenz von 1100 ms bleibt. Ab einer Latenz von 200 ms fangen die Darstellungsmodelle an, sich voneinander abzuheben, und teilen sich in zwei Gruppen auf. Die erste Gruppe besteht aus den einzigen beiden Darstellungsmodellen, die Time Compensation aktiviert haben: „DIS-DR erster Ordnung“ mit Snap und linearer Konvergenz. Diese Gruppe erzeugt bei einer Latenz von 200 bis 850 ms einen wesentlich kleineren Positionsfehler als die zweite Gruppe. Die Darstellungsmodelle der zweiten Gruppe, bestehend aus allen verbleibenden Dead Reckoning Darstellungsmodellen, die keine Time Compensation verwenden, bleiben über die gesamte Erhebung untereinander nahezu unverändert. Ab einer Latenz von 850 ms überschneiden sich die Gruppen, wobei die zweite Gruppe nun den geringsten Positionsfehler liefert, während sich die erste Gruppe rapide verschlechtert. Zu erkennen ist, dass schon der geringste durchschnittliche Positionsfehler bei einer Latenz von 800 ms, unter Verwendung von „DIS-DR erster Ordnung, mit Snap-Konvergenz und Time Compensation“, ca. 20 Pixel beträgt. Dies ist ein Wert, der die Realisierung von Arcade-Action-Spielen unter GPRS unmöglich machen dürfte. Aufgrund dessen konzentriert sich

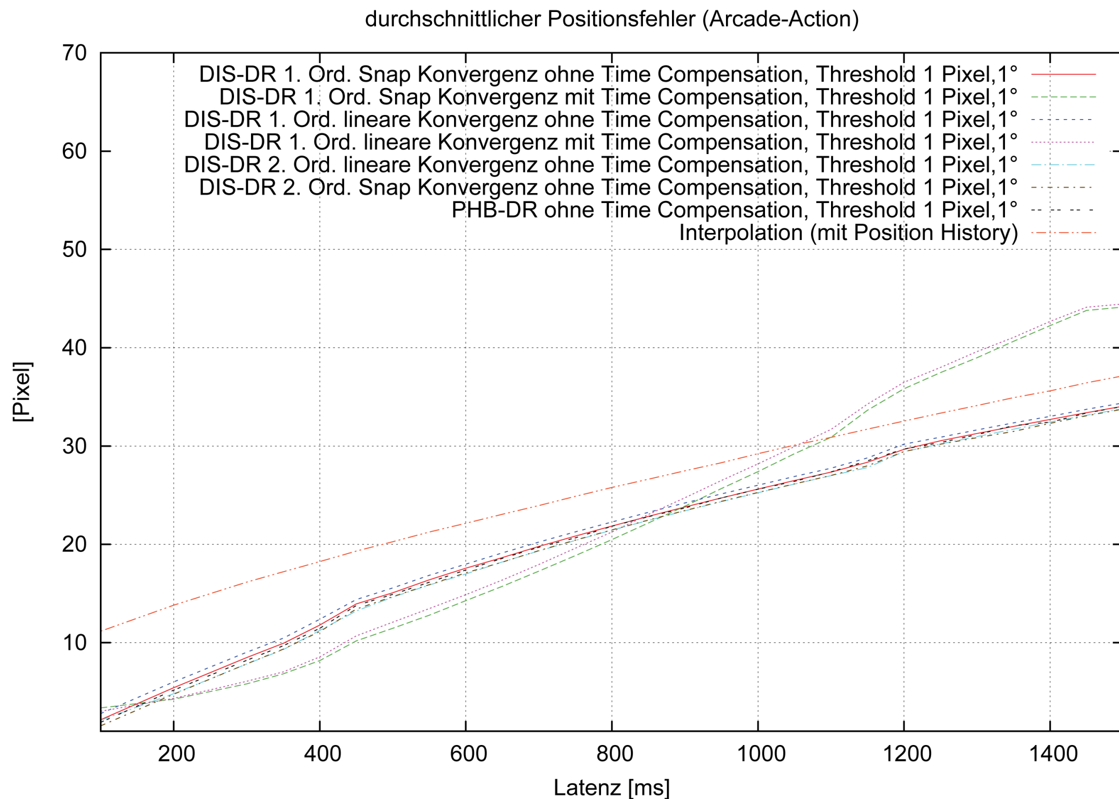


Abbildung 3.40: durchschnittliche Abweichung zur tatsächlichen Position (Arcade-Action)

die folgende Auswertung auf den für UMTS-Verbindungen relevanten Latenzbereich von bis zu 400 ms. Weiterhin wurden, für eine klarere Unterscheidung der Verläufe, in Abb. 3.41 und Abb. 3.42 die Darstellungsmodelle „DIS-DR erster Ordnung mit Snap-Konvergenz und linearer Konvergenz“ entfernt. „DIS-DR zweiter Ordnung“ weist einen sehr ähnlichen Verlauf zu diesen Darstellungsmodellen auf, und erzeugt insgesamt etwas geringere Positionsfehler. Des weiteren wurde „Interpolation“ nicht mehr aufgeführt. Nach Abbildung 3.41 sind die für Arcade-Action-Spiele unter UTMS vermeintlich besten Darstellungsmodelle „DIS-DR erster Ordnung, aktivierte Time Compensation“ mit Snap oder linearer Konvergenz. In der Tat liefern diese beiden Modelle den kleinsten durchschnittlichen Positionsfehler, in dem für UMTS relevanten Bereich von 200 bis 400 ms. Betrachtet man jedoch den maximalen Positionsfehler der einzelnen Messschritte (Abb. 3.42) wird schnell klar, dass die Positionsgenauigkeit dieser Modelle sehr stark schwankt. „DIS-DR erster Ordnung mit Snap-Konvergenz und Time Compensation“ erzeugt schon bei geringer Latenz derart extreme Schwankungen, das es praxisuntauglich ist. Der maximale Positionsfehler von „DIS-DR erster Ordnung, mit linearer Konvergenz und Time Compensation“ beträgt durchschnittlich das 1,9-fache von „DIS-DR zweiter Ordnung, mit Snap-Konvergenz“, dem am besten abschneidenden Darstellungsmo-

dell. Der durchschnittliche Positionsfehler ist jedoch im Mittel, in dem für UMTS relevanten Bereich von 200 bis 400 ms, nur um das 0,8-fache besser. Bei einer Verwendung von „DIS-DR erster Ordnung, mit linearer Konvergenz und Time Compensation“ würde also für eine Verbesserung des durchschnittlichen Positionsfehlers um 1,8 Pixel eine Erhöhung des maximalen Fehlers um durchschnittlich 10,9 Pixel in Kauf genommen werden. Die Darstellungsmodelle, die also für Arcade-Action-Spiele unter UMTS den besten Kompromiss zwischen durchschnittlichem und maximalem Positionsfehler liefern, sind daher „DIS-DR zweiter Ordnung, deaktivierte Time Compensation“ mit Snap oder linearer Konvergenz. Es bleibt jedoch zu untersuchen, inwiefern sich die Unverhältnismäßigkeit bei Modellen mit aktivierter Time Compensation zwischen der Verbesserung des durchschnittlichen Positionsfehlers und der Verschlechterung des maximalen Positionsfehlers mit einer Limitierung der Time Compensation relativieren lässt. Eventuell sind die Darstellungsmodelle „DIS-DR erster Ordnung, mit Snap oder linearer Konvergenz“ zusammen mit einer limitierten Time Compensation eine mögliche Alternative.

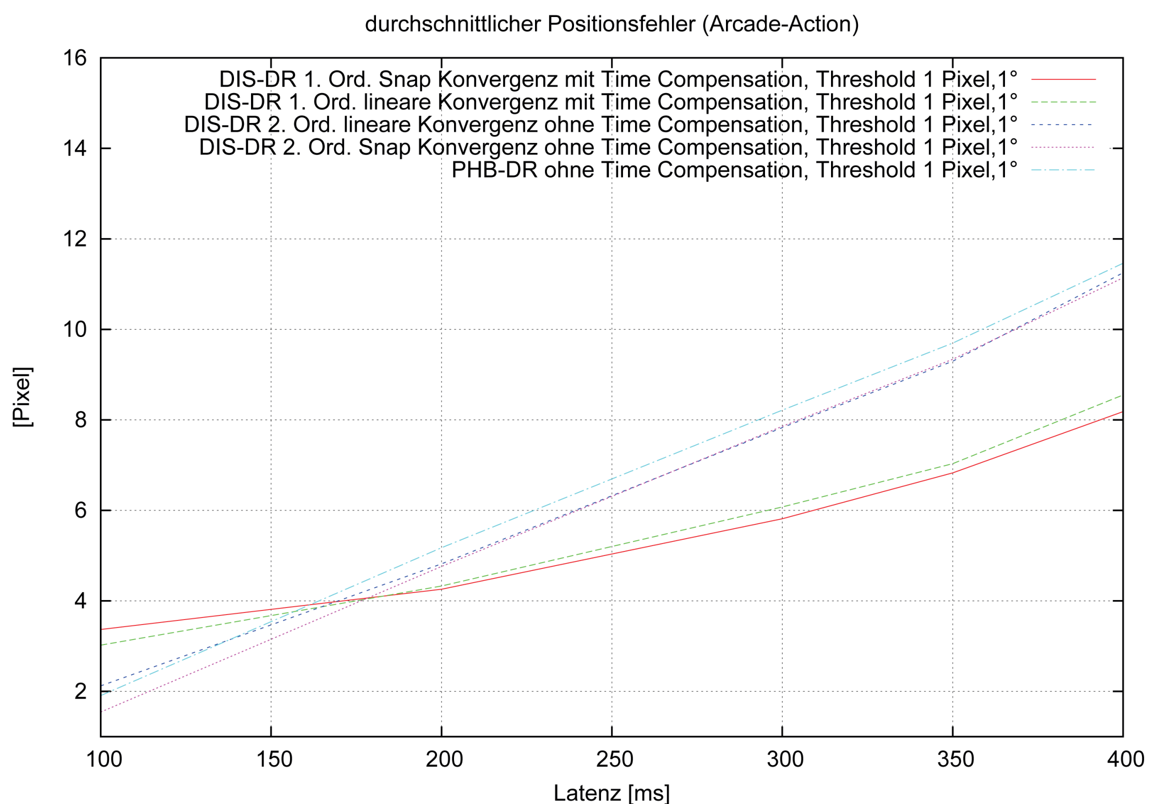


Abbildung 3.41: durchschnittliche Abweichung zur tatsächlichen Position (Arcade-Action)

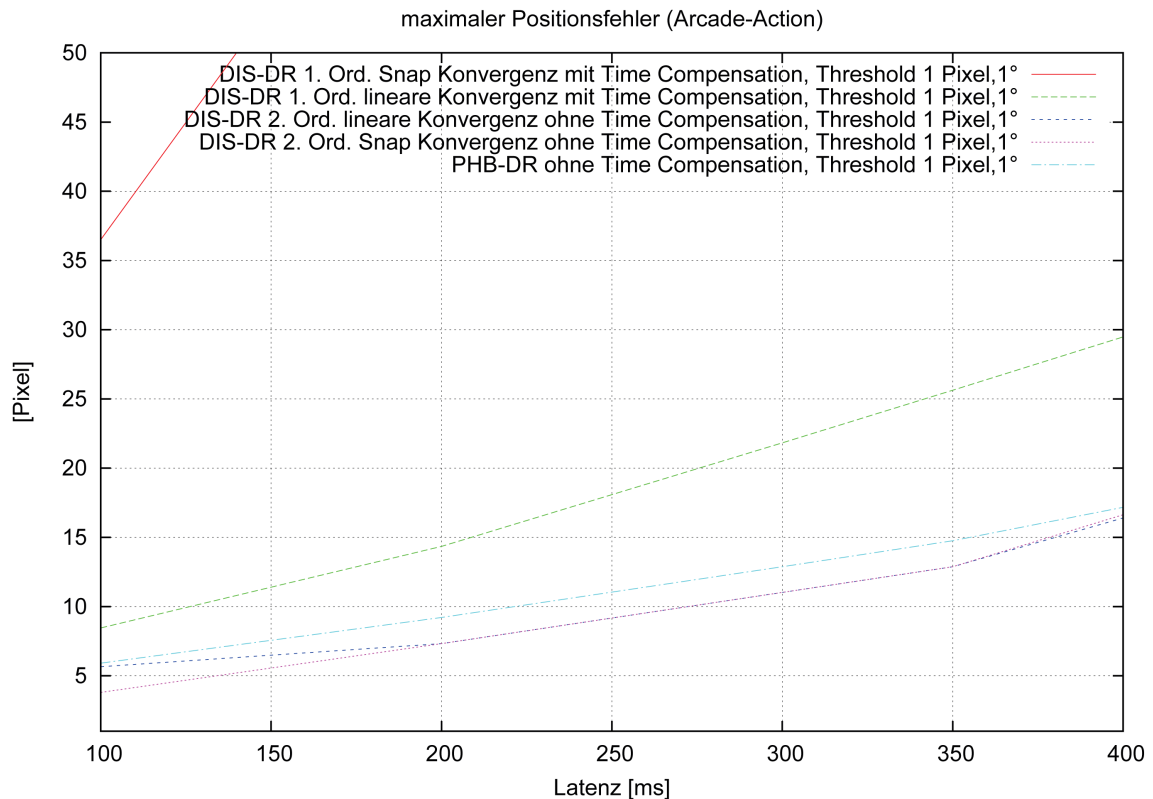


Abbildung 3.42: maximale Abweichung zur tatsächlichen Position (Arcade-Action)

Fazit Arcade-Action

Die Auswertung des Positionsfehlers hat ergeben, dass „DIS-DR erster Ordnung, mit Snap-Konvergenz und Time Compensation“ die Technik ist, die den kleinsten Positionsfehler bei Arcade-Action-Spielen unter GPRS erzeugt. Unter UMTS sind „DIS-DR zweiter Ordnung, deaktivierte Time Compensation, mit Snap oder linearer Konvergenz“ die Techniken, welche die größte Positionsgenauigkeit der entfernten Entität sicherstellen. Dabei sorgt ein linearer Konvergenzalgorithmus für eine etwas bessere Benotung des Bewegungsverlaufs als Snap, und ist daher das Modell der Wahl. Damit keine unrealistischen Rotationen der Entität entstehen, sollte die Orientierungskonvergezzeit auf 1 ms gesetzt werden. Es bleibt noch zu untersuchen, inwiefern sich die unrealistische diagonale Bewegung der entfernten Entität, die bei linearer Konvergenz entsteht, auf das Spielgefühl auswirkt.

Arcade-Action unter GPRS

Da sich schon der geringste durchschnittliche Positionsfehler auf das 2-fache der Entitätsgrösse beläuft, sind Arcade-Action-Spiele unter GPRS nicht möglich. Ähnlich zu Sportspie-

len ist ein genaues Abstimmen der eigenen Bewegung im Hinblick auf die Bewegung der anderen Entitäten nötig. So werden bei Bomberman Bomben in Gänge platziert, die nach einer Zeit explodieren. Um den Gegner mit der Explosion zu erwischen, muss seine Bewegung genau beobachtet, und die Bombe im richtigen Moment platziert werden. Bei einem hohen Positionsfehler befindet sich der Gegner aber schon längst in einem anderen Gang und damit in Sicherheit, und wird nicht wie erwartet von der Explosion getroffen. Auch dürften die bei Bomberman häufig auftretenden Kollisionen sehr irritierend sein, wenn die Entitäten dabei durchschnittlich um das 2-fache ihrer Größe neu positioniert werden.

Arcade-Action unter UMTS

Von denen für Arcade-Action-Spiele unter UTMS empfohlenen Techniken wurde „DIS-DR zweiter Ordnung, deaktivierte Time Compensation, mit linearer Konvergenz“ für den abschließenden Test unter realen Netzwerkbedingungen (siehe Tabelle 2.4.5) ausgewählt. Die Verwendung von linearer Konvergenz weist einen besseren Bewegungsverlauf auf als Snap-Konvergenz. Da sich während der Tests zeigte, dass Techniken mit limitierter Time Compensation eine mögliche Alternative sind, wurde ein weiterer Vergleich mit „DIS-DR erste Ordnung, lineare Konvergenz“ und einer auf 100 ms limitierten Time Compensation durchgeführt:

DIS-DR Darstellungsmodell:	2. Ord., ohne TC, lin. Konv.	1. Ord., TC 100, lin. Konv.
Bewegungsverlauf:	gut (2)	sehr gut (1,5)
durchschnittl. Senderate:	0,444 kByte/s	0,365 kByte/s
maximale Senderate:	0,852 kByte/s	0,734 kByte/s
durchschnittl. Positionsfehler:	10,073 Pixel	9,073 Pixel
maximaler Positionsfehler:	26,29 Pixel	21,24

Es stellt sich heraus, dass „DIS-DR erster Ordnung, mit linearer Konvergenz“ und einer auf 100 ms limitierten Time Compensation in der Tat das bessere Modell ist. Gegenüber „DIS-DR zweiter Ordnung, deaktivierte Time Compensation, mit linearer Konvergenz“ fallen der durchschnittliche sowie der maximale Positionsfehler kleiner aus, die Senderate ist niedriger, und der Bewegungsverlauf wurde als „sehr gut“ statt nur „gut“ empfunden. Bei UMTS ist der maximale Positionsfehler gerade mal so groß wie der durchschnittliche Positionsfehler bei GPRS. Dennoch ist es fraglich, ob eine durchschnittliche Positionsabweichung um das 0,9-fache und eine maximale Positionsabweichung um das 2-fache der Entitätsgrösse noch innerhalb der Grenzen des Akzeptablen liegen, um Kollisionen fair zu erkennen. Die befürchteten diagonalen Bewegungen bei Verwendung von linearer Konvergenz traten beim Test unter realen Netzwerkbedingungen allerdings nur bei Paketverlust wahrnehmbar auf. Bei einer Paketverlusrate von 1% (siehe Tabelle 2.4.5) geschieht dies äußert selten und ist

somit hinnehmbar.
Auch bei der Auswertung des Arcade-Action-Presets stellte sich also heraus, dass das am besten abschneidende Darstellungsmodell wahrscheinlich nicht genügt, um ein optimales Spielerlebniss unter UMTS zu ermöglichen. Durch die Verwendung von Local Lag oder eine Verringerung der Geschwindigkeit der Entitäten bestehen jedoch Möglichkeiten, den Positionsfehler weiter zu verringern, um eventuell ein akzeptables Spielerlebniss zu erreichen.

Zu empfehlendes Darstellungsmodell für Arcade-Action unter UMTS:

Darstellungsmodell:	DIS-DR 1.Ordnung mit Time Compensation, begrenzt auf 100 ms
Threshold Position:	10% der Größe des Spielobjekts (1 Pixel)
Threshold Orientierung:	1°
Konvergenzmodell:	Lineare Konv. mit Konv.zeit: 100 ms Position, 1 ms Orientierung

3.3.5 Fazit der Auswertung

Alle vorhergehenden Tests haben ergeben, dass „DIS-DR erster Ordnung mit linearer Kovergenz und begrenzter Time Compensation“ das für alle untersuchten Spielegenres empfehlenswerteste Darstellungsmodell ist. Die Parameter „Time Compensation“ und „Konvergenzzeit“ sind dabei für Genres mit einer trägeren Entität (Simulation) höher eingestellt als bei Genres mit bedeutend agilerer Entität (Sportsgame, 3D-Shooter, Arcade-Action). „DIS-DR zweiter Ordnung“ schnitt größtenteils beim Vergleich des Positionsfehlers ähnlich gut ab wie „DIS-DR erster Ordnung“, stand bei der Bewertung der Bewegung jedoch durchgehend schlechter da. Auch „PHB-DR“ konnte sich nicht gegen „DIS-DR erster Ordnung“ durchsetzen, und lieferte immer etwas größere Positionsfehler oder schlechtere Bewegungsverläufe. Daher lässt sich darauf schließen, dass Darstellungsmodelle, welche die Beschleunigung der Entität berücksichtigen, einerseits zwar auch für geringe Positionsfehler sorgen können, andererseits meistens ein schlechteres Bewegungsverhalten der entfernten Entität verursachen. „Interpolation“ hingegen liefert immer eine nahezu perfekte Wiedergabe des Bewegungsverlaufs, ist aber das Darstellungsmodell, welches bei allen Spielegenres den größten Positionsfehler verursacht. Ob sich dies durch eine Verminderung des Updateintervalls relativieren lässt, bleibt für weitere Untersuchungen offen.

3.4 Evaluation des Realtime-Simulators

Wir haben in unserer Testapplikation „Realtime-Simulator“ alle gängigen Techniken implementiert, die bis heute zur Darstellung von entfernten Spielobjekten in verteilten virtuellen Welten genutzt werden. Diese Spielobjekte können durch zahlreiche Parameter sehr exakt an das Steuerungsverhalten unterschiedlichster Spielgenres angepasst werden. Um die

Tauglichkeit der Techniken unter den besonderen Bedingungen der mobilen Netze testen zu können, können die Parameter Latenz und Paketverlust offline simuliert werden. Zudem ist es möglich, den Datenverkehr über die Neutron Realtime Library zu lenken, um eine Konfiguration auf echten Handys in verschiedenen Netzen zu testen.

Einschränkungen

- Momentan sind im Realtime-Simulator nur Sitzungen mit maximal zwei Spielern möglich. Unter Verwendung der Neutron Realtime Library könnte die Applikation aber recht schnell erweitert werden, um theoretisch beliebig viele Spieler zu unterstützen.
  Dabei ergibt sich jedoch ein weiteres Problem: die Implementierung der Latenzsimulation erlaubt nur eine Simulation der Gesamtverzögerung der Kommunikation zwischen Clients. Dies hat zur Folge, dass bei Spielsitzungen von mehr als zwei Spielern bei allen Spielern die gleiche Latenz erzwungen wird. Für eine Analyse von Spielen mit mehr als zwei Spielern wäre es jedoch realistischer, dass die Spieler verschiedene Latenzen haben.

- Scrolling wird noch nicht unterstützt, weshalb Simulationen mit sehr langen Beschleunigungsphasen momentan nicht gut getestet werden können, da die Entitäten den sichtbaren Bildschirmbereich verlassen.

- Kollisionen und Projektilwaffen werden von unserer Applikation nicht simuliert. Dazu müsste eine autoritative Komponente implementiert werden, die Kollisionen erkennt und entsprechende Events an alle Teilnehmer verschickt. Die Tauglichkeit der Darstellungsmodelle für diese Spielfeatures können wir zwar in Zahlen über den durchschnittlichen und maximalen Positionsfehler erfassen, aber wie stark diese Fehler den Spielspaß und die Fairness tatsächlich beeinflussen, lässt sich in dieser Version des Realtime-Simulators nur ungefähr abschätzen.

Erweiterungsmöglichkeiten

Ein mögliches Problem ist, dass unsere Implementation Fließkommazahlen benutzt, die in J2ME-kompatiblen Handys erst ab dem CLDC1.1 Standard unterstützt werden. Dieser Standard wird von vielen Herstellern noch eher stiefmütterlich behandelt, und ist oft durch relativ langsame Softwarealgorithmen implementiert. Die meisten Entwickler greifen deshalb auf performante Fixed-Point-Math-Bibliotheken wie z. B. *MathFP* [6] zurück, die spezielle Methoden zur Berechnung von Fließkommazahlen bereitstellen, die letztendlich in ganzen Zahlen vom Typ *integer* repräsentiert werden. Methoden, die eine Schnittstelle zur Neutron-Realtime-Library darstellen und Fließkommazahlen als Argumente übernehmen, sollten des-

[6] http://mywebpages.comcast.net/ohommes/MathFP/mathfp_bg.html

halb getrennte Argumente vom typ integer für Vor- und Nachkommastelle bereitstellen, um Konflikte zu vermeiden. Entsprechend sollte die Neutron-Bibliothek intern auf die Nutzung einer performanten Fixed-Point-Bibliothek umgerüstet werden, um auch CLDC1.0 konforme Endgeräte zu unterstützen.
Um die Nutzung des Realtime-Simulators als Showcase zugänglicher zu machen, wäre es wünschenswert, die simple Visualisierung der Entitäten als Dreiecke durch repräsentativere Grafiken auszutauschen. Die Entitäten könnten beispielsweise beim Sportsgame-Preset als Fußballspieler oder beim Simulations-Preset als Raumschiff visualisiert werden. Dies würde auch für eine subjektive Einschätzung der Einsatzfähigkeit der verschiedenen Darstellungsmodelle förderlich sein.
Nicht zuletzt ist zu beachten, dass unsere Applikation nur mit 2D-Vektoren arbeitet. Außerdem wurde keine Winkelbeschleunigung implementiert, stattdessen drehen sich die Entitäten nur mit konstanter Geschwindigkeit.

Anwendungsmöglichkeiten

Der gegenwärtige Zustand des Realtime-Simulators erfüllt Exit Games' Bedarf eines Showcase, der das Verhalten verschiedener Spielgenres unter Verwendung von gängigen Latenzausgleichstechniken visualisiert und auswertbar macht. Anhand des Realtime-Simulators können sich potentielle Kunden schnell einen ungefähren Überblick verschaffen, wie sich die Entitäten ihrer Echtzeit-Multiplayerspiele voraussichtlich in Mobilfunknetzwerken verhalten werden. Sinnvoll wäre es daher, den Realtime-Simulator in Form einer auf Handys und Emulatoren ausführbaren JAR-Datei oder als Java-Applet im WWW zur Verfügung zu stellen.

3.5 Erweiterung der Neutron-Echtzeit-Plattform

Auf Basis unserer Untersuchungen sind mehrere Erweiterungsmöglichkeiten für die Neutron-Echtzeit-Plattform denkbar. Wir stellen dabei zuerst die Komponenten vor, die wir als am praktikabelsten ansehen.

3.5.1 Kapselung der Darstellungsmodelle

Um die schnelle und unkomplizierte Umsetzung einer Spielidee zu ermöglichen, bietet sich eine Integration der von uns evaluierten Darstellungsmodelle in die Neutron-Client-Bibliothek an. Um Entwicklern die Einarbeitung in die internen Mechanismen zu ersparen, sollten dabei auch Out-of-the-Box-Lösungen für typische Anwendungsfälle angeboten werden. Abbildung 3.43 zeigt das Klassendiagramm unseres Erweiterungsvorschlags.

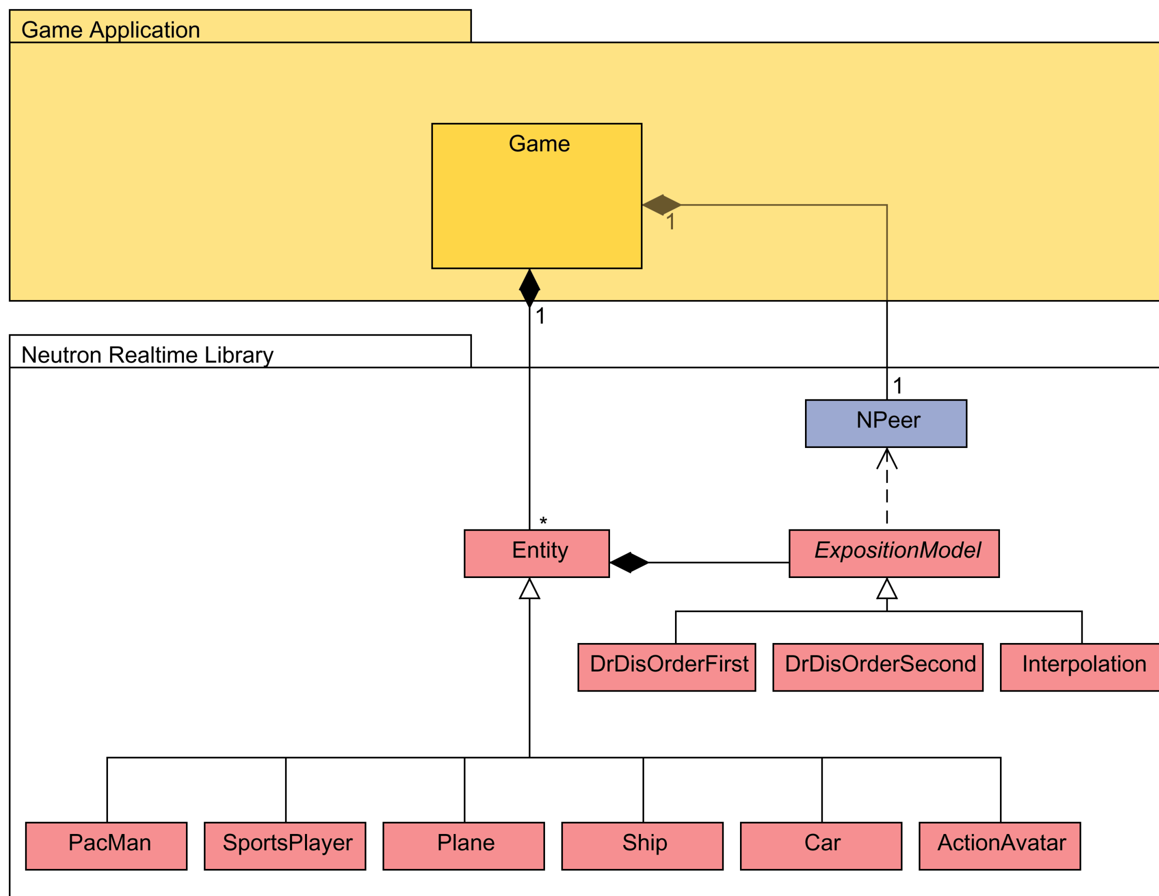


Abbildung 3.43: Entwurf zur Ergänzung der Neutron-Client-Bibliothek

Um die von uns implementierten Algorithmen nutzen zu können, müssen die relevanten Spielobjekte von der vorgefertigten Oberklasse *Entity* erben. Entity enthält entsprechend Abbildung 3.4 grundlegende Instanzvariablen zu Position, Geschwindkeit und Ausrichtung der Entität. Entity bildet die zentrale Schnittstelle zur Konfiguration sämtlicher entitätsbezogener Eigenschaften, inklusive des genutzten Darstellungs- und Konvergenzmodells. Auf diesem Weg werden die Implementierungsdetails vor dem Anwender verborgen. Um im typischen Anwendungsfall möglichst wenig Einstellungen vornehmen zu müssen, stellen wir mehrere Presets zu typischen Spielobjekten bereit, wie z.B. PacMan (Arcade), Flugzeug (Simulation) oder ein Avatar für 3D-Rollen- und Actionspiele. Ein Entwickler kann von diesen vorgefertigten Einstellungen dasjenige wählen, das dem von ihm gewünschten Spielobjekt bzw.-genre am ehesten entspricht. Nach Auswahl des Presets sollte die grundlegende Funktionalität schon gegeben sein, wobei auch automatisch die von uns jeweils empfohlenen Darstellungs- und Konvergenzmodelle mit einer geeigneten Konfiguration initialisiert werden. Der Entwickler kann mit Hilfe unserer Testapplikation diese Einstellungen danach

weiter optimieren.
In unserer Darstellung wurden diese Presets als Unterklassen von Entity realisiert, von denen ein Anwendungsentwickler beim Erstellen eigener Spielobjektklassen erben kann. In der Praxis können die Presets auch als Intialisierungsmethoden oder als Flags im Konstruktor aufgerufen werden, da Unterklassen in J2ME unverhältnismäßig viel Speicherplatz beanspruchen, und die Neutron-Bibliothek möglichst schlank gehalten werden sollte. Der Enwickler muss in jedem Gameloop-Durchlauf die cycle-Methode jeder Entity aufrufen, wobei als Argument die Dauer des letzten Durchlaufs übergeben wird. Die Funktionsweise entspricht dem bereits in 3.2 erläuterten Ablauf. In wieweit das Steuerungsmodell ebenfalls gekapselt werden kann oder sollte ist fraglich, da es eng mit dem von uns genutzten physikalischen Modell gekoppelt ist, das für viele Anwendungen entweder zu einfach oder auch schon zu komplex sein wird. Als alternative Schnittstelle zum Darstellungsmodell könnte auch je eine setter-Methode für den Geschwindigkeits- und den Beschleunigungs-Vektor eingerichtet werden, der im aktuellen Frame auf die Entität wirkt. Der Entwickler kann das zu Grunde liegende Steuerungsmodell dann so einfach oder so komplex gestalten wie nötig, allerdings geht dabei dann auch die Out-of-the-Box-Funktionalität verloren.

Vorteile:

- Darstellungsmodell wird gekapselt
- Out-of-the-Box-Lösung für typische Anwendungsfälle möglich
- keine Erweiterung der Serverkomponente erforderlich
- keine zusätzliche Serverlast

3.5.2 Serverkomponenten

Im folgenden sind Techniken aufgeführt, die sich für die Realisierung von verschiedenen Echtzeitspielen als notwendig herausgestellt haben und deren Funktionalität modular, in Form einer optionalen Serverkomponente, gekapselt werden kann:

3.5.2.1 Kollision von Entitäten

Um konsistente Kollisionen zwischen Entitäten zu ermöglichen, ist eine autoritative Instanz nötig, die Kollisionen erkennt und an alle Spielteilnehmer kommuniziert (siehe Abschnitt 2.5.2). Optimalerweise sollte diese Aufgabe der Neutron-Server übernehmen, da der Server zum einen als Zentrum der Kommunikation die Kollisionen zeitlich vor allen Clients erkennen kann, und sich zum anderen clientseitige Darstellungsfehler nicht auf die Spiellogik auswirken. Eine Realisierung setzt allerdings voraus, dass die betroffenen Entitäten das von

uns entwickelte Framework der Entity-Klasse und ihrer Darstellungmodelle übernehmen, da der Server zur Interpretation der Events die gemeinsamen Serialisierungsprotokolle und Algorithmen benutzen muss.
Zur eigentlichen Kollisionserkennung existieren verschiedene Ansätze mit unterschiedlicher Komplexität, vom einfachen Überprüfen auf Überschneidung der rechteckigen Entitätsgrenzen („Hitboxen"), bis hin zum exakten Überschneidungsvergleich einzelner Polygone. Wird eine Kollision erkannt, teilt die Kollisionskomponente allen Spielern mittels eines speziellen Events mit, an welcher Position und zu welchem Zeitpunkt und mit welcher Geschwindigkeit die Kollision stattgefunden hat. Da der Neutron-Server weder eine Kenntnis der Spiellogik hat, noch ein Modell der Spielumgebung besitzt, kann der Server in diesem Bereich keine weitere Verantwortung übernehmen. Für die Berechnung der aus Kollisionen resultierenden Auswirkungen sind in diesem generischen Ansatz deshalb die Entitäten zuständig. Die Unkenntnis über die Spielumgebung führt auch dazu, dass die Modellierung von Projektilobjekten nicht vom Server übernommen werden kann, da der Server nie wissen kann, wann das Projektil mit der Umgebung kollidiert. Dementsprechend ist auch die Kollisionserkennung zwischen Projektilen und der von Spielern gesteuerten Entitäten serverseitig ausgeschlossen, und muss ebenfalls von den Clients durchgeführt werden.

Vorteile:

- erhöhter Grad an Konsistenz durch schnelle und zentrale Erkennung von Kollisionen
- keine serverseitige Kenntnis der Spielumgebung nötig

Nachteile:

- zusätzliche Serverlast
- das im vorherigen Absatz vorgestellte Entity-Framework muss genutzt werden

3.5.2.2 Interface für Levelstrukturen

Die bisher vorgestellten Erweiterungsmöglichkeiten lassen sich generisch für viele Echtzeitspiele nutzen, ohne die Server-Logik an ein spezielles Spiel anpassen zu müssen. Um dem Server weiterführende Verantwortung übertragen zu können, müsste er Kenntnis von der Struktur der Spielumgebung besitzen. Um ein Einbinden der Levelstruktur in die Server-Komponente zu ermöglichen, wäre also eine Schnittstelle nötig, die das Einlesen von Maps realisiert. Es wäre dabei sinnvoll, die in der Spieleentwicklung von gebräuchlichen Mapeditoren generierten Mapformate zu unterstützen. Wie praktikabel dieser Ansatz ist, hängt davon ab, ob in der Praxis bei der Enwicklung von Multiplayerspielen auf standardisierte Formate zurückgegriffen wird.

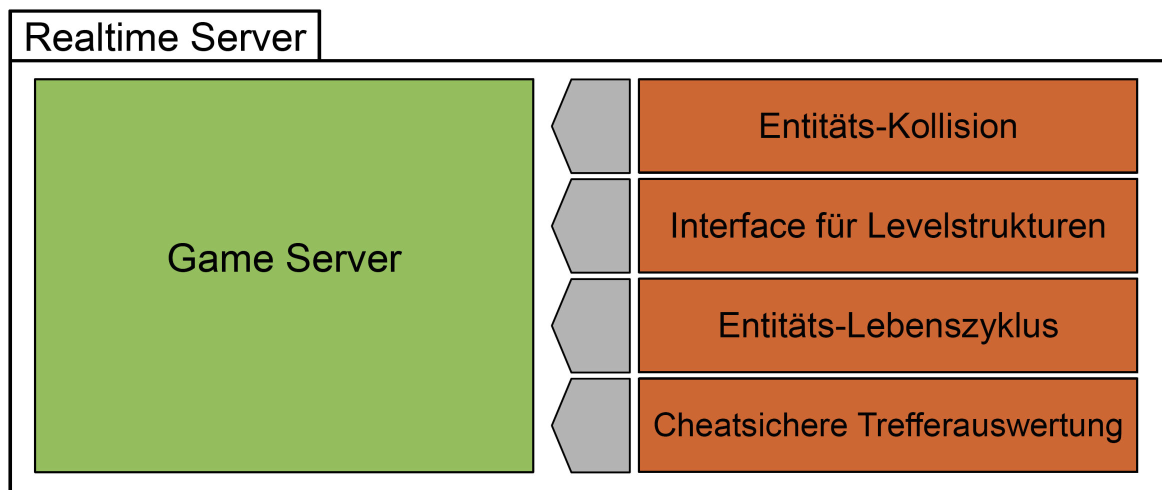


Abbildung 3.44: Erweiterung des Realtime Servers

3.5.2.3 Lebenszyklus der Entitäten

Genauso sinnvoll wie eine Kollisions-Komponente ist eine Komponente, die den Lebenszyklus der Entitäten verwaltet. Wie schon besprochen ist eine clientseitig implementierte Logik, die über das Ableben von Entitäten entscheidet, äußerst cheatanfällig. Des weiteren ließe sich diese auch wieder nicht zufriedenstellend als verteilten Algorithmus realisieren. Durch eine dritte Instanz, welche für die Konsistenzerhaltung beim Ableben von Entitäten sorgt, würde sich auch das in 2.5.2 besprochene Problem der Ereigniskausalität beheben lassen. Konkret ließe sich die Lebenszykluskomponente durch eine Erweiterung unseres Systems der Darstellungsmodelle realisieren. Mit einem zusätzlichen Attribut „Hitpoints“ in der Klasse Entity und einem weiteren Klassenmodell für Waffen, welches den jeweils verursachten Schaden der Waffen verwaltet, wäre es der Lebenszykluskomponente möglich festzustellen, ob eine Entität bei Beschuss getötet wurde. Darüber hinaus müsste die Lebenszykluskomponente die „Local Rollback“ Timewarptechnik (siehe Abschnitt 2.5.3) implementieren. Innerhalb einer definierbaren Zeitspanne ist eine Änderung der zu propagierenden Nachricht aufgrund neuer Informationen über den kausalen Zusammenhang möglich. Falls also Spieler B Spieler C erschießt, die Lebenszykluskomponente jedoch innerhalb der definierten Zeitspanne (beispielsweise 100 ms) eine Nachricht von Spieler A erhält und erkennt, dass A aufgrund des absoluten Zeitstempels eigentlich vorher B erschoss, wird letztendlich an die Clients nur die Nachricht „A erschießt B“ versendet. Die Nachricht „B erschießt C“ wird also nicht propagiert, und die Konsistenz ist auf allen Clients gewahrt. Dies führt allerdings dazu, dass bei jedem Schuss die Latenz für eine Antwort erhöht wird, da ja um eine zusätzliche Zeitspanne gewartet wird, dass weitere, die Kausalität beeinflussende Ereignisse auszuschließen sind. Ein Phänomen, das im Zusammenhang mit Local Rollback

entsteht, ist, dass es zwar ein sofortiges visuelles Feedback des Beschusses gibt, was vermuten lässt, dass der Gegner tödlich getroffen wurde, stattdessen aber die eigene Entität einen Moment später stirbt. Auch kann es passieren, dass das vorher eingetretene Ereignis nach Überschreitung der Zeitspanne ankommt und keine Beachtung mehr findet. Dies lässt sich nicht verhindern, und muss für eine Erhaltung der Konsistenz in Kauf genommen werden. Die Zeitspanne muss jedoch so gewählt werden, dass diese Fehler möglichst selten provoziert werden und trotzdem eine möglichst geringe Verzögerung der Antwort besteht. Die mit der Neutron Realtime Library entstehenden typischen Latenzen betragen bei UMTS ca. 170 ms. Dies sollte also die für Echtzeitspiele in mobilen Netzen übliche Zeitspanne bei Local Rollback sein, um den auf weitere kausalitätsbeeinflussende Ereignisse gewartet wird. Ob die dadurch entstehende RTT von ca. 510 ms noch einen akzeptablen Wert darstellt, der die Interaktivität des Spiels nicht allzu negativ beeinflusst, bleibt zu untersuchen. Die Erweiterung des Realtime Servers um eine Komponente, die den Lebenszyklus von Entitäten verwaltet, ist in Abb. 3.44 dargestellt.

3.5.2.4 Cheatsichere Trefferauswertung

Eine weitere Erweiterungsmöglichkeit besteht im Implementieren von Lag Compensation gemäß 2.8.2, um das unsichere Konzept der Treffer-Events durch eine cheatsichere serverseitige Trefferauswertung von Instant-Hit-Waffen zu ersetzen. Neutron-Spiele, die ohne proprietäre Server-Komponente auskommen müssen, können aber niemals cheatsicher gemacht werden, da ihre Spiellogik zwangsläufig auf die Clients ausgelagert werden muss. Eine Implementierung von Lag Compensation bietet sich deshalb in Form eines Moduls an, um die die Game Server Komponente bei Bedarf für spezielle Spiele erweitert werden kann.

Vorteile:

- stark erhöhte Cheatsicherheit durch serverseitige Trefferauswertung
- Spielbarkeit wird durch Berücksichtigung der clientseitigen Sicht gewahrt

Nachteile:

- Levelstruktur muss bekannt sein
- Entity-Framework muss genutzt werden
- zusätzliche Serverlast

Kapitel 4

Zusammenfassung und Ausblick

Wir haben in unserer Arbeit verschiedene Probleme untersucht, die beim Realisieren von Multiplayer-Echtzeitspielen in verteilten Systemen auftreten. Dabei wurden die erschwerten Bedingungen in aktuellen Mobilfunknetzen berücksichtigt. Das Hauptproblem war hierbei die hohe Latenz, die je nach Spieldesign entweder zu starken Inkonsistenzen oder zu einer schlechten Spielbarkeit führen kann. Ein zentrales Problem war diesbezüglich insbesondere die lokale Darstellung entfernt gesteuerter Spielobjekte im Hinblick auf die Verzögerung der Positionsangaben. Unterschiedlich hohe Latenzen der Mitspieler können auch die Fairness des Spielablaufs beeinträchtigen, da die zeitliche Reihenfolge der Ereignisse verfälscht wird.
Wir haben im Entwurfsteil dieser Arbeit verschiedene Softwaretechniken implementiert, die in aktuellen kommerziellen PC-Spielen und in militärischen Simulationen zum Latenzausgleich angewandt werden. Zur Evaluation dieser Techniken haben wir in unserer Testumgebung die drei QoS-Eigenschaften Latenz, Paketverlust und Varianz (Jitter) simuliert, die in aktuellen Mobilfunknetzen typischerweise vorherrschen. Um bei der Evaluierung verschiedene Spielegenres berücksichtigen zu können, wurde das Steuerungsverhalten stark parametrisiert und typische Einstellungen zu fest definierten „Presets“ zusammengefasst. Dadurch war es uns am Ende unserer Testläufe möglich, konkrete Vorschläge zur Realisierung bestimmter Spielegenres zu machen. Allgemein konnten wir bei der Darstellung entfernter Entitäten in den meisten Fällen „Dead Reckoning 1. Ordnung“ empfehlen, einer Technik, die neben der Position auch die Geschwindigkeit, nicht aber die Beschleunigung der Entität überträgt. Wir haben insgesamt festgestellt, dass unter dem zur Zeit performantesten Mobilfunkstandard UMTS auch actionlastige Echtzeitspiele realisierbar sind, wenn auch mit einigen Einschränkungen. Je träger das Beschleunigungsverhalten der Spielobjekte ist, desto besser kann trotz Latenz die Konsistenz über alle Teilnehmer hinweg gewahrt werden. Durch die zunehmende Verbreitung des Java-Standards MIDP2.0 lassen sich heute aber auch unter dem langsameren Mobilfunk-Standard GPRS viele echtzeitorientierte Multiplayerspiele verwirklichen.
Wir haben schließlich auf Basis unserer Untersuchungen ein grobes Konzept mit verschiedenen Komponenten erarbeitet, um die die Neutron-Echtzeit-Plattform der Firma Exit Games

langfristig erweitert werden könnte. Die Zeitsynchronisation bildet dabei die Grundvoraussetzung für viele Latenzausgleichstechniken, und wurde während der Erstellung dieser Arbeit bereits implementiert.
Abschließend stellt sich die Frage, welche der untersuchten Spielegenres sich in den nächsten Jahren auf dem Handy etablieren könnten. Multiplayer-Rennspiele im Stil von „Trackmania“ eignen sich aufgrund der kurzen Spielphasen und des offenen Spielverlaufs sehr gut für eine Handy-Portierung, auch wenn der Fahrspaß je nach Handymodell oft durch die unergonomische Tastatur getrübt wird. Teambasierte 3D-Actionspiele werden zwar wahrscheinlich ebenfalls bald auf Handys umgesetzt, bieten aber unserer Meinung nach mehr Prestigewert für den Entwickler als langfristigen Spielspaß für den Nutzer, weil die Steuerung zu stark unter den eingeschränkten Eingabemöglichkeiten leidet. Für actionorientierte Multiplayer-Rollenspiele sehen wir gute Marktchancen, auch wenn das Chatting auf Handys eher mühsam und zeitraubend ist, und der soziale Aspekt dieses Genres dadurch in den Hintergrund gerät. Voice-over-IP wird als Alternative von Rollenspielern im Allgemeinen abgelehnt, da dieses Genre stark von der spielerischen Immersion lebt, die durch VoIP schnell zunichte gemacht wird. Insgesamt sind wir skeptisch, ob komplexere handybasierte Echtzeit-Multiplayerspiele jemals eine höhere Verbreitung erlangen werden als Singleplayer-Handyspiele. Das Handy eignet sich als Spieleplattform in erster Linie zur Überbrückung kurzer Wartezeiten, weshalb die unter Umständen langwierige Spielinitialisierung und die oft ungewisse Spieldauer eine Hemmschwelle darstellen. Ein laufendes Spiel kann oft nicht mehr ohne Weiteres abgebrochen werden, da man Gefahr läuft, seine Mitspieler zu verärgern oder seine Online-Statistiken zu verschlechtern. Ein weiterer Faktor sind die anfallenden Verbindungskosten, da Flatrate-Tarife im Mobilfunkbereich momentan noch relativ teuer sind. Allgemein bekommt das Handy als Spieleplattform für Gelegenheitsspieler heute zunehmend Konkurrenz durch die immer professionelleren Internetbrowser-Spiele, und für passionierte Hardcorespieler könnte in den nächsten Jahren wiederum ein handliches Notebook mit UMTS-Karte zum Standard werden, auf dem sie ihr Lieblingsspiel auch unterwegs ohne Qualitätseinbußen spielen können. Ein echter Mehrwert entsteht langfristig für handybasierte Multiplayerspiele also hauptsächlich, wenn dabei auch der mobile Charakter des Handys genutzt wird, beispielsweise im Zusammenhang mit der Berücksichtung der aktuellen Position. Die notwendigen Schnittstellen zur Realisierung dieser „Location Based“-Spiele sind bereits Teil des Java-Standards, und mit GPS existiert auch eine Technologie, deren Auflösung hoch genug ist, um sinnvoll mit den vorgestellten Echtzeitalgorithmen kombiniert werden zu können. Die neuen Handystandards bieten also durchaus Potential für das Umsetzen wirklich neuartiger Spielideen, wie auch der Gewinner des diesjährigen *International Mobile Gaming Awards*[7] namens *Triangler* beweist. Ob sich solche Spiele vom eher akademischen Experiment zu einem echten Marktsegment entwickeln werden, bleibt abzuwarten. Gleiches gilt für die kommende Generation der Smartphones mit Multitouch Display:

[7] www.imgawards.com

Es muss sich noch zeigen, ob die neue Interfacetechnologie den Tastaturen der heutigen Handygeneration überlegen ist.
Eine Weiterentwicklung unserer Testapplikation wird zunächst von Exit Games als Showcase für die Neutron-Echtzeit-Plattform genutzt, und soll schließlich auch Entwicklern zur Verfügung gestellt werden. Wir hoffen, dass unsere Arbeit dabei hilft, wesentliche Hindernisse bei der Realisierung verteilter Echtzeitspiele aus dem Weg zu räumen.

Literaturverzeichnis

[Aggarwal u. a. 2004] AGGARWAL, Sudhir ; BANAVAR, Hemant ; KHANDELWAL, Amit ; MUKHERJEE, Sarit ; RANGARAJAN, Sampath: Accuracy in dead-reckoning based distributed multi-player games. In: *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2004, S. 161–165. – ISBN 1-58113-942-X

[Bailey 1982] BAILEY, R.W.: *Human Performance Engineering: A Guide for System Designers*. Prentice Hall, 1982

[Beatson 1986] BEATSON, R K.: On the convergence of some cubic spline interpolation schemes. In: *SIAM J. Numer. Anal.* 23 (1986), Nr. 4, S. 903–912. – ISSN 0036-1429

[Beigbeder u. a. 2004a] BEIGBEDER, Tom ; COUGHLAN, Rory ; LUSHER, Corey ; PLUNKETT, John ; AGU, Emmanuel ; CLAYPOOL, Mark: The effects of loss and latency on user performance in unreal tournament 2003. In: *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2004, S. 144–151. – ISBN 1-58113-942-X

[Beigbeder u. a. 2004b] BEIGBEDER, Tom ; COUGHLAN, Rory ; LUSHER, Corey ; PLUNKETT, John ; AGU, Emmanuel ; CLAYPOOL, Mark: The effects of loss and latency on user performance in unreal tournament 2003®. In: *NetGames '04: Proceedings of 3rd ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2004, S. 144–151. – ISBN 1-58113-942-X

[Bernier 2001] BERNIER, Yahn W.: Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. In: *Game Developers Conference* (2001)

[Bettner und Terrano 2001] BETTNER, P. ; TERRANO, M.: 1500 archers on a 28.8: Network programming in age of empires and beyond / Game Developers Conference 2001. March 2001. – Forschungsbericht

[Cai u. a. 1999] CAI, Wentong ; LEE, Francis B. S. ; CHEN, L.: An auto-adaptive dead reckoning algorithm for distributed interactive simulation. In: *PADS '99: Proceedings of the*

thirteenth workshop on Parallel and distributed simulation. Washington, DC, USA : IEEE Computer Society, 1999, S. 82–89. – ISBN 0-7695-0155-9

[Chandler und Finney 2005] CHANDLER, Angie ; FINNEY, Joe: On the effects of loose causal consistency in mobile multiplayer games. In: *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2005, S. 1–11. – ISBN 1-59593-156-2

[Claypool u. a. 2006] CLAYPOOL, Mark ; KINICKI, Robert ; LEE, William ; LI, Mingzhe ; RATNER, Gregory: Characterization by measurement of a CDMA 1x EVDO network. In: *WICON '06: Proceedings of the 2nd annual international workshop on Wireless internet*. New York, NY, USA : ACM Press, 2006, S. 2. – ISBN 1-59593-510-X

[Conner und Holden 1997a] CONNER, Brook ; HOLDEN, Loring: Providing a low latency user experience in a high latency application. In: *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 1997, S. 45–ff.. – ISBN 0-89791-884-3

[Conner und Holden 1997b] CONNER, Brook ; HOLDEN, Loring: Providing a low latency user experience in a high latency application. In: *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*. New York, NY, USA : ACM Press, 1997, S. 45–ff.. – ISBN 0-89791-884-3

[Cristian 1989] CRISTIAN, Flaviu: Probabilistic Clock Synchronization. In: *Distributed Computing* 3 (1989), Nr. 3, S. 146–158

[Diot und Gautier 1999] DIOT, C. ; GAUTIER, L.: a distributed architecture for multi-player interactive applications on the internet. In: *IEEE Network Magazine* 13 (1999), July/August, Nr. 4

[Duncan und Gracanin 2003] DUNCAN, Thomas P. ; GRACANIN, Denis: Algorithms and analyses: pre-reckoning algorithm for distributed virtual environments. In: *WSC '03: Proceedings of the 35th conference on Winter simulation*, Winter Simulation Conference, 2003, S. 1086–1093. – ISBN 0-7803-8132-7

[Fraser 2000] FRASER, John: *Zeroping Mutator for Unreal Tournament*. http://zeroping.home.att.net/faq.html. 2000

[Fujimoto 2000a] FUJIMOTO, Richard M.: Comparing optimistic and conservative synchronisation. In: *Parallel and Distributed Simulation Systems* (2000), S. 172

[Fujimoto 2000b] FUJIMOTO, Richard M.: Push Algorithms. In: *Parallel and Distributed Simulation Systems* (2000), S. 272

[Gautier und Diot 1998] GAUTIER, L. ; DIOT, C.: Design and Evaluation of MiMaze, a Multi-Player Game on the Internet. In: *icmcs* 00 (1998), S. 233. ISBN 0-8186-8557-3

[Henderson 2003] HENDERSON, T.: *The effects of relative delay on networked games*. London, UK, University of London, Masterarbeit, April 2003. – http://www.cs.dartmouth.edu/ tristan/pubs/thesis.pdf

[Heng 2006] HENG, Stefan: Breitbandige Mobilfunktechnologie UMTS ist Realität / Deutsche Bank Research. URL http://www.dbresearch.de/PROD/DBR_INTERNET_DE-PROD/PROD0000000000198071.pdf, 2006. – Forschungsbericht

[Holma und Toskala 2006] HOLMA, Harri ; TOSKALA, Antti: *HSDPA /HSUPA for UMTS*. John Wiley, 2006. – ISBN 978-0-470-01884-2

[idsoftware 1997] IDSOFTWARE: *Doom Source Code*. 1997. – URL http://www.doomworld.com/ports/linux_unix.shtml

[idSoftware 1999] IDSOFTWARE: *Quake Source Code*. 1999. – URL ftp://ftp.idsoftware.com/idstuff/source/q1source.zip

[Standards Committee on Interactive Simulation (SCIS) of the IEEE Computer Society 1996] IEEE COMPUTER SOCIETY, USA Standards Committee on Interactive Simulation (SCIS) of the: *IEEE standard for distributed interactive simulation communication services and profiles*. IEEE Std 1278.2. 1996

[Ishibashi und Tasaka 2003] ISHIBASHI, Yutaka ; TASAKA, Shuji: Causality and media synchronization control for networked multimedia games: centralized versus distributed. In: *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2003, S. 42–51. – ISBN 1-58113-734-6

[J. Gundermann 2004] J. GUNDERMANN, A. S.: Mobilfunknetze - von 2G nach 3G. UMTS, GPRS, GSM, WLAN, Medien Institut Bremen, 2004. – ISBN 978-3932229732

[J.Chesterfield 2005] J.CHESTERFIELD, J.Crowcroft: Experiences with multimedia streaming over 2.5G and 3G Networks. 2005. – Forschungsbericht

[Lambert 2001] LAMBERT, Mike: *Player's Guide to UT Netcode*. 2001. – URL http://www.nub-clan.de/_.php3?url=netguide/show.php3

[Lamport 1978] LAMPORT, Leslie: Time, clocks, and the ordering of events in a distributed system. In: *Commun. ACM* 21 (1978), Nr. 7, S. 558–565. – ISSN 0001-0782

[Lincroft 1999] LINCROFT, Peter: *The Internet Sucks: Or, What I Learned Coding X-Wing vs. TIE Fighter*. 1999. – URL http://www.gamasutra.com/features/19990903/lincroft_01.htm

[LLP 2006] LLP, PricewaterhouseCoopers: *PricewaterhouseCoopers LLP Global Entertainment and Media Outlook: 2006-2010*. http://www.pwc.com/Extweb/industry.nsf/docid/8CF0A9E084894A5A85256CE8006E19ED. 2006

[M. Mauve und Effelsberg 2004] M. MAUVE, V. H. ; EFFELSBERG, W.: Local-Lag and Timewarp: Providing Consistency for Replicated Continuous Applications. In: *IEEE Transactions on Multimedia* 6 (2004), Feb, Nr. 1

[Miller und Thorpe 1995] MILLER, D.C. ; THORPE, J. A.: SIMNET: The advent of simulator networking. In: *Proceedings of the IEEE83* (1995), S. 1114–1123

[Mills 1992] MILLS, David L.: *RFC 1305: Network Time Protocol (Version 3) Specification, Implementation*. März 1992. – URL ftp://ftp.internic.net/rfc/rfc1305.txt

[mindcontrol] MINDCONTROL: *NAT punch-through*. – URL http://www.mindcontrol.org/~hplus/nat-punch.html

[Nichols und Claypool 2004] NICHOLS, James ; CLAYPOOL, Mark: The effects of latency on online madden NFL football. In: *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA : ACM Press, 2004, S. 146–151. – ISBN 1-58113-801-6

[Salzman 2002] SALZMAN, Lee: *enet*. 2002. – URL http://enet.cubik.org/

[Schiller 2003] SCHILLER, Jochen: *Mobilkommunikation*. 2., überarb. Aufl. München [u.a.] : Pearson Studium, 2003. – 565 S. : Ill., graph. Darst. S. – URL http://bibserv21.bib.uni-mannheim.de:8080/openedu/library/opac/library.opac.html?query=nd=102171971. – ISBN ISBN 3-8273-7060-4

[Sheldon u. a. 2003] SHELDON, Nathan ; GIRARD, Eric ; BORG, Seth ; CLAYPOOL, Mark ; AGU, Emmanuel: The effect of latency on user performance in Warcraft III. In: *NetGames '03: Proceedings of the 2nd workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2003, S. 3–14. – ISBN 1-58113-734-6

[Shneiderman 1984] SHNEIDERMAN, Ben: Response Time and Display Rate in Human Performance with Computers. In: *ACM Comput. Surv.* 16 (1984), Nr. 3, S. 265–285

[Singhal und Cheriton 1994] SINGHAL, Sandeep K. ; CHERITON, David R.: Using a Position History-Based Protocol for Distributed Object Visualization. Stanford, CA, USA : Stanford University, 1994. – Forschungsbericht

[Timm-Giel 2004] TIMM-GIEL, Andreas: *UMTS/GPRS Performance Measurementsand Evaluation*. ITG FG521-Workshop TU Hamburg-Harburg. February 2004. – URL http://www.comnets.uni-bremen.de/itg/itgfg521/aktuelles/fg-treffen-200204/ITG_FG521_Hamburg_200204_TimmGeil.pdf

[Yasui u. a. 2005] YASUI, Takahiro ; ISHIBASHI, Yutaka ; IKEDO, Tomohito: Influences of network latency and packet loss on consistency in networked racing games. In: *NetGames '05: Proceedings of 4th ACM SIGCOMM workshop on Network and system support for games*. New York, NY, USA : ACM Press, 2005, S. 1–8. – ISBN 1-59593-156-2

[Zhang und Martin 1997] ZHANG, Zhimin ; MARTIN, Clyde F.: Convergence and Gibbs' phenomenon in cubic spline interpolation of discontinuous functions. In: *J. Comput. Appl. Math.* 87 (1997), Nr. 2, S. 359–371. – ISSN 0377-0427

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Juli 2007 Marcel Köhler, Andreas Pongs