Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Nicolai Fabisch

Konzeption und Realisierung eines performanten Verkehrsnetzes für den Einsatz in Massively Multiplayer Online Realtime Strategy Games

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Nicolai Fabisch

Konzeption und Realisierung eines performanten Verkehrsnetzes für den Einsatz in Massively Multiplayer Online Realtime Strategy Games

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Olaf Zukunft
Zweitgutachter : Prof. Dr. Thomas Thiel-Clemen

Abgegeben am 10. November 2012

Nicolai Fabisch

Thema der Bachelorarbeit

Konzeption und Realisierung eines performanten Verkehrsnetzes für den Einsatz in Massively Multiplayer Online Realtime Strategy Games.

Stichworte

Verkehrsnetz, Online Games, MMORTS, Ereignisorientierte Simulation

Kurzzusammenfassung

Diese Arbeit beschreibt die Konzeption und Entwicklung eines performanten Verkehrsnetzes für den Einsatz in MMORTS, lässt sich aber auch in anderen Spielen einsetzen. Durch den Einsatz ereignisorientierter Simulation, Graphentheorie und Vorberechnung, wird die durch Kollisionsabfragen verursachte Grundlast auf das System entfernt. Die in der Arbeit durchgeführten Performanz-Tests geben Aufschluss über die Leistungsfähigkeit der konkreten Implementierung.

Nicolai Fabisch

Title of the paper

Design and implementation of a performant traffic system for use in Massively Multiplayer Online Realtime Strategy Games.

Keywords

Traffic system, online games, MMORTS, discrete event simulation

Abstract

This thesis describes the development and implementation of a performant traffic system for the use in MMORTS, but may also be used in other games. The concept uses discrete event simulation, graph theory and pre-calculation to get rid of the base load caused by collision querys. The test results give an insight into the performance of the specific implementation presented in this paper.

Inhaltsverzeichnis

Abkürzungsverzeichnis

- **RPG**
  Role playing game; Ein Rollenspiel
- **RTS**
  Real-time strategy (game); Ein Echtzeit Strategiespiel
- **FPS**
  First-person shooter; Ein Kampfspiel aus der Egoperspektive
- **MMOG**
  Massively multiplerer online game; Ein Massen-Mehrspieler-Online-Gemeinschaftsspiel
- **MMORTS**
  Massively multiplayer online real-time strategy (game); Ein Massen-Mehrspieler-Online-Echtzeit-Strategiespiel
- **MMORPG**
  Massively multiplayer online role-playing game; Ein Massen-Mehrspieler-Online-Rollenspiel
- **LLRB-Tree**
  Left-leaning red-black tree; Nach links tendierender Rot-Schwarz Baum.

1 Einleitung

Strategiespiele nehmen an den verkauften Computerspielen nach [ESA12] über 25% ein und sind damit das meistverkaufte Genre unter den Computerspielen der Mitglieder der Entertainment Software Association [ESA] im Jahr 2012. Im Bereich der *„Massively multiplerer online game*s“ (MMOs) spielen sie derzeit keine Rolle. Dieser Spieltyp wird von Rollenspielen dominiert [WikiMMOList]. Zwar sind Strategiespiele vereinzelt auch zu finden, allerdings handelt es sich fast ausschließlich um Browsergames.

1.1 Motivation

Echtzeit-Strategiespiele scheinen den Sprung in das neue Genre *„Massively multiplayer online real-time strategy game“ (*MMORTS) noch nicht gemeistert zu haben. In den letzten Jahren haben sich eher Browsergames im Strategiegenre (Happy Farm [WikiHappyFarm], Travian [Travian]) durchgesetzt, welche eine eingeschränkte Darstellungsvielfalt aufweisen und den Grundcharakter der Spiele stark anpassen, um diese im Browser spielbar zu machen. Sie sind dafür serverseitig sehr viel einfacher zu verwalten und sind von nahezu jedem Endgerät aus abrufbar.

Ein Problem der Echtzeit-Strategiespiele ist vor allem die hohe Komplexität [Smith1]. Ein Teilnehmer eines MMORTS benötigt in der Regel auf den Spieleservern wesentlich mehr Ressourcen als ein Teilnehmer eines MMORPG oder gar eines Browsergames. Der hohe Grad an Komplexität und das Zusammenspiel vieler Systeme macht den Betrieb von RTS im Multiplayer Modus bereits bei sehr wenigen Teilnehmern zu einer großen Herausforderung. Bei MMOs kommt erschwerend ein hohes wirtschaftliches Risiko für den Betreiber durch die Serververwaltung hinzu.

Doch der Reiz an einem Echtzeit-Strategiespiel besteht gerade darin, ein komplexes System in begrenzter Zeit zu verwalten. Es kann sich dabei um die Verwaltung von Kriegsarmeen handeln, die Gestaltung eines Freizeitparks oder die Verwaltung einer ganzen Stadt. In nahezu allen Spielen dieser Art ist die Förderung und Verarbeitung von Rohstoffen ein Teil des Spielkonzepts. In einigen Spielen, wie in den bekannten Spielereihen Siedler oder Anno, basiert das ganze Spielkonzept auf diesen Wertschöpfungsketten. Eine besondere Bedeutung hat dabei der Transport der Waren. Die Förderung, Verarbeitung und der Handel mit Ressourcen und Waren zwischen den Produktionsgebäuden und Einheiten vieler verschiedener Spieler in einem riesigen dynamischen Wirtschaftsraum stellt ein faszinierendes Szenario dar und ist die Hauptmotivation dieser Bachelorarbeit.

1.2 Zielsetzung

In dieser Arbeit soll ein Konzept für ein ressourcensparendes Verkehrsnetz präsentiert werden, welches auch für den Einsatz in MMORTS geeignet ist. Dabei soll versucht werden, das Spielerlebnis des Spielers, die Wirtschaftlichkeit des Betreibers und das

Verbrauchsverhalten in Einklang zu bringen. In diesem Zusammenhang soll ein Spiel-Prototyp entwickelt werden, in dem das Verkehrsnetz präsentiert und getestet werden kann. Es wird der Anspruch erhoben, den Spiel-Prototyp in einer Weise zu gestalten, der eine Weiterentwicklung begünstigt. Die eingesetzten Techniken für die Verwaltung der Verkehrsnetze in den serverseitigen Spielwelten sollen erläutert werden. Weiterhin soll die Arbeit grundlegende Forschungsergebnisse liefern, die bei einer Weiterentwicklung des in der Motivation geschilderten Szenarios verwendet werden können.

1.3 Abgrenzung

Ziel der Arbeit ist keine vollständige Software oder Spiel-Software. Die Arbeit erhebt auch nicht den Anspruch, eine multiplayerfähige Software zu entwickeln. Die Ausarbeitung dient hier vor allem der Ideen-präsentation und soll einen alternativen Ansatz zur üblichen Ressourcenverwaltung heutiger Spiele bieten. Die Arbeit orientiert sich vor allem an der Performanceoptimierung und der Integration des entwickelten Verkehrsnetzes in ein mögliches Serversystem. Das in dieser Arbeit entwickelte Verkehrsnetz erhebt auch keinen Anspruch darauf, ein System aus der Realität genau abzubilden. Das Verkehrsnetz soll lediglich einem Spiel die Möglichkeit bieten, mit einem geringen Verbrauchsverhalten (nach ISO/IEC 9126 [ISO]) möglichst viele Objekte zu bewegen.

1.4 Gliederung der Arbeit

In Kapitel ***2*** werden grundlegende, für diese Arbeit relevante, Begrifflichkeiten erläutert. Hierzu gehört beispielsweise eine Beschreibung von MMOs, Strategiespielen, Simulationen, Kollisionsmanagement und diskreten Event-Systemen.

Im Anschluss folgt ein Anwendungsszenario in Kapitel ***3***, welches einen möglichen Einsatz des Verkehrsnetzes aufzeigt. Das Anwendungsszenario wird anschließend in einer vereinfachten Form beschrieben und in dieser Arbeit verwendet.

Es folgt die Analyse in Kapitel ***4***. Hier werden bestehende Spielsysteme, soweit möglich, analysiert und Erkenntnisse zusammengetragen. Dabei sind vor allem die von Spieleentwicklern berichteten Erfahrungen interessant. Diese beeinflussen maßgeblich die Konzeptbildung der Arbeit. Der zweite Teil des vierten Kapitels enthält eine Bewertung der gewonnenen Erkenntnisse, sowie die Aufstellung von Anforderungen an den Spieleprototyp und des Verkehrsnetzes.

In Kapitel ***5*** folgt die Konzeption. Hier werden die Anforderungen aus dem Anwendungsszenario aufgegriffen und mit den Erkenntnissen aus der Analyse kombiniert. Das dabei entstehende Konzept für den Spiel-Prototyp dient als Grundlage für das Konzept des Verkehrsnetzes, welches im Anschluss erläutert wird.

Schließlich folgt in Kapitel ***6*** die Realisierung. Hier wird das Konzept in ein lauffähiges System umgewandelt, welches den Anspruch hat, die Grundzüge des Konzepts darzustellen und einen Einblick in die Verwaltungstechnik zu schaffen.

Weiterhin soll das System erlauben, grundlegende Aussagen über die Performanz des Systems zu treffen. Diese Tests folgen in Kapitel ***7***, in welchem auch die Beschreibungen sonstiger Systemtests zu finden sind.

Eine Bewertung der Umsetzung, anhand der Anforderungen, erfolgt in Kapitel ***8***, bevor in Kapitel ***9*** eine Zusammenfassung der Arbeit folgt und ein Ausblick über zukünftige Erweiterungen gegeben wird.

2 Grundlagen

In diesem Kapitel werden die für die Arbeit relevanten Themenbereiche erläutert. Es wird auf Computerspiele und deren Zusammenhang mit Simulationen, sowie auf andere relevante Themenbereiche eingegangen. Die folgenden Unterpunkte stehen nicht in direkten Zusammenhang oder bauen auf einander auf.

2.1 Massively Multiplayer Online Games

In einem Massively Multiplayer Online Game (MMOG) können viele Teilnehmer zur gleichen Zeit über das Internet am Spielgeschehen teilnehmen. Dabei bleiben alle Errungenschaften eines Spielers erhalten, auch wenn er das Spielgeschehen temporär verlässt. In der Regel kommunizieren und interagieren die Teilnehmer miteinander (Kampf, Handel, Kooperatives Interagieren usw.) [Hal08 S4], [Gre09 S23].

Prinzipiell ist es möglich alle gängigen Spiele-Genres mit den Konzepten der MMOGs zu kombinieren. An dieser Stelle wird nur auf die aktuell drei wichtigsten Genres eingegangen [ESA12], [Gre09 S24]:

MMORPG (Massively multiplerer online role-playing game), die Kombination aus RPGs (Rollenspiele) und MMOs

MMORTS (Massively multiplerer online realtime strategy game), die Kombination aus RTSs (Strategiespiele) und MMOs

MMOFPS (Massively multiplerer online first person shooter), die Kombination aus FPSs (First-Person-Shooter) und MMOs

Hervorzuheben sind hier die MMORPGs. Diese kombinieren die Elemente von Rollenspielen mit einer Onlinewelt, welche rund um die Uhr erreichbar ist. Die MMORPGs haben in den letzten Jahren einen enormen Bekanntheitsgrad erlangt und prägen aktuell den Begriff der MMOs. In den anderen Genres gibt es aktuell noch keine vergleichbar erfolgreichen Spieletitel [WikiMMO]. Verwiesen sei hier noch auf die aktuellen Entwicklungen im Bereich der MMOGs. Im Bereich der MMOFPS befindet sich Planetside 2 in der Alpha-Phase und verspricht bis zu 1500 Spieler auf einer Spielkarte [Planetside]. Bei den MMORTS wurde das Spiel End of Nations angekündigt, bei dem bis zu 50 Spieler auf einer Karte interagieren sollen [EndOfNations].

Die Verwaltung der Spielerdaten und anderer Informationen übernehmen bei MMOS zentrale Server. Nach [Ale05 S341] werden *„Alle kommerziellen MMOGs heutzutage in der Client/Server-Architektur erstellt“*. Der Client dient nur als Präsentationseinheit für den Teilnehmer. Weiterhin leitet er die Eingaben des Teilnehmers an den Server weiter, wobei dieser entscheidet, ob die Eingaben valide sind. Die serverseitige Verwaltung von Nutzerinformationen kann auf verschiedene Arten umgesetzt sein und ist für diese Arbeit nicht entscheidend.

2.2 Echtzeit Strategiespiele

In Echtzeit Strategiespielen läuft die Zeit im Gegensatz zu rundenbasierten Strategiespielen kontinuierlich ab. Alle Teilnehmer (Computerspieler oder reale Spieler) können ihre Handlungen zur gleichen Zeit ausführen. Die Teilnehmer versuchen den Sieg über andere Teilnehmer durch Planung zu erlangen, dabei muss der Teilnehmer wirtschaftliche, taktische und strategische Handlungen ausführen. In der Regel überwiegen die kriegerischen Handlungen gegenüber den Mitstreitern das Spielgeschehen. Der wirtschaftliche Teil ist gering und teilweise nicht existent. Es gibt zudem Strategiespiele, in denen nicht notwendigerweise über Kampfhandlungen interagiert werden muss. Es kann beispielsweise versucht werden, über spezielle Siegbedingungen zu gewinnen oder über Diplomatie den Mitstreiter zu besiegen[Ada06 S469 ff.].

Die Definition von Echtzeit Strategiespielen variiert stark. Eine Vielzahl von Spielen kann je nach Definition in das Genre einbezogen oder ausgeschlossen werden. Es sei hier auf diverse Auslegungsarten in der Literatur verwiesen [Ada06 S469 ff.] [Gre09 S21 f.], bei denen die Definition an einigen Stellen offen gelassen wird [Ada06 S472]. Dies gilt auch für die Einordnung von existierenden Spielen, welche nicht klar nach einer Definition geschieht. Beispielhaft ist die Liste der MMORTSs von Wikipedia zu nennen [WikiRTSList], in der sich mehrere Titel befinden [WikiHappyFarm], die unter einer strengen Definition, wie nach [Adams] oder [WikiRTS], nicht in das Genre gehören. In dieser Arbeit wird eine sehr allgemeine Definition, die auch in [Adams] zu finden ist, verwendet, vor allem um die breite Einsatzmöglichkeit des im späteren Verlauf entstehenden Verkehrsnetzes zu beschreiben.

[...] Blizzard's Rob Pardo who explained an RTS as simply "A strategic game in which the primary mode of play is in a real-time setting." […] [Adams]

Frei übersetzt: *„Ein Echtzeit Strategiespiel ist [...] ein Strategiespiel, in welchem der primäre Spielmodus in Echtzeit stattfindet. […]“*

2.3 Wirtschaftssimulationen (Spiel)

In Wirtschaftssimulationen muss ein Teilnehmer das Management und die Konstruktion von Objekten beauftragen und überwachen. Er muss dabei ökonomisch sinnvoll handeln, um erfolgreich zu sein. In den meisten Spielen dieser Art ist das Ziel, eine Art von ‚Imperium‘ aufzubauen. In Aufbauspielen wie SimCity [WikiSimCity] wäre dies beispielsweise eine Metropole, in Geschäftsspielen (Business Simulation) eine hohe Marktpräsenz.

Es existieren zudem Hybriden, die nicht klar in Strategiespiele oder Wirtschaftssimulationen einzuteilen sind, da sie stark von wirtschaftlichen Aspekten abhängen aber über kriegerische Handlungen gegenüber den Mitstreitern gewonnen werden. Beispielhaft sei hier die Anno-Reihe genannt [WikiAnno]. So lange es sich bei dem jeweiligen Spiel um ein Echtzeit-Spiel handelt, werden auch Wirtschaftssimulationen in die Definition von Echtzeit-Strategiespielen

(siehe Abschnitt ***2.2***) einbezogen, da man sowohl militärisch als auch wirtschaftlich strategisch vorgehen kann [Ada06 S469 ff.].

2.4 Cheaten und Hacking

Der Betrug (engl. Cheaten; Ausnutzen eines Fehlers im Spiel zum eigenen Vorteil) sowie das ‚Hacking' (Mutwilliges Zerstören oder Manipulieren von Informationen) ist in Multiplayer Spielen ein großes Problem [Sal03 S280]. Dabei nutzt ein Teilnehmer Techniken aus, die in der Form nicht vorgesehen sind und erlangt so einen Vorteil gegenüber seinen Mitstreitern. Diese Techniken können Fehler in der Spielmechanik sein, Drittprogramme, die dem Spieler mehr Informationen zur Verfügung stellen, oder auch Methoden, über die er unerlaubt Fähigkeiten oder Ressourcen erlangt. Die Verfolgung von Cheatern und Hackern kostet sehr viel Zeit und Ressourcen, ist aber nötig, da durch diese unfairen Methoden das Spielerlebnis aller Spieler stark beeinträchtigt wird. Dabei können die ausgenutzten Fehler zum Teil enorme Ausmaße annehmen und die Spielwelt und -wirtschaft großflächig beeinträchtigen. Der asiatische Server für das Spiel Diablo 3 musste beispielsweise auf Grund eines Spielfehlers, der es ermöglichte Spielgegenstände sehr billig zu erstellen, abgeschaltet und auf einen früheren Datenbankzustand zurück versetzt werden [Diablo31] [Diablo32]. Diablo 3 besitzt ein Auktionshaus, in welchen Spielgegenstände für echtes Geld getauscht werden können. Bei diesem tausch geht ein erheblicher Teil des Geldes an den Betreiber Blizzard Entertainment selbst[Diablo33]. Das sehr einfache Generieren von Spielgegenständen hat in diesem Fall einen direkten Einfluss auf den Umsatz des Unternehmens, da bei einer erhöhten Anzahl an begehrten Gegenständen die Auktionspreise fallen. Nach [Sal03 S280] kann die Auseinandersetzung mit Cheatern und Hackern die Hälfte aller zu Verfügung stehenden Ressourcen der Wartung und Weiterentwicklung kosten. [Ale05 S307 ff.] bietet zu diesem Thema weiterführende Informationen.

2.5 Kollisionserkennung

Die Kollisionserkennung ist ein fundamentales Problem in der Computersimulation. Ziel ist die Erkennung von Kollisionen zwischen verschiedenen (beweglichen) Objekten. Dabei soll sich beispielsweise ein Objekt nicht durch ein anderes hindurch bewegen, sondern ähnlich den physikalischen Gesetzen zurückgeworfen oder gestoppt werden. Diese Kalkulationen sind teilweise sehr komplex und rechenintensiv und können im schlimmsten Fall einen Laufzeitaufwand von $O(n^2)$ nach der O-Notation [WikiONotation] besitzen [Gre09 S622 ff.] [Eri04 S24]. Dabei muss ein beträchtlicher Teil der Prozessor-Leistung aufgewendet werden, welcher dann anderen Bereichen nicht mehr zur Verfügung steht. Eine schnelle und kostengünstige Kollisionserkennung ist daher ein essentielles Ziel von virtuellen Umgebungen. Dies gilt im Besonderen für Echtzeit-Computerspiele, da hier nur wenig Zeit für jeden Berechnungszyklus der Spielwelt zur Verfügung steht [Eri04 S25 ff.].

2.6 Diskrete Simulation

In der Informatik versteht man unter einer Simulation die *„Nachbildung von Vorgängen auf einer Rechneranlage auf der Basis von Modellen“* [Cla06]. Es existieren mehrere Unterkategorien einer Simulation[WikiComputersimulation]. In dieser Arbeit werden nur diskrete Simulationen behandelt. Dabei finden Zustandsänderungen in der diskreten Simulation sprunghaft und zu diskreten Zeitpunkten statt. Sie werden durch Ereignisse in der Simulation ausgelöst. Die Ereignisse bestimmen den folgenden Systemzustand. Der Wechsel von einem Zustand zum nächsten kann durch feste Zeitintervalle oder Ereignisse in der Simulationsumgebung stattfinden [Mat89 S201 ff.].

2.6.1 Ereignisgesteuerte Simulation

In der ereignisgesteuerten Simulation finden zu bestimmten Simulationszeitpunkten Ereignisse statt. Die Simulations-Kalkulation springt dabei von Ereignis zu Ereignis. Die reale Zeit spielt keine Rolle. Finden demnach nur wenige Ereignisse mit wenig Rechenaufwand statt, so schreitet die Simulationszeit schnell voran. Finden viele Ereignisse mit hohem Rechenaufwand statt, verläuft die Simulationszeit in Relation zur Echtzeit langsam [Mat89 S201 f.].

In [DES S560] wird der formale Ablauf einer diskreten ereignisgesteuerten Simulation erläutert.

2.6.2 Zeitgesteuerte Simulation

Bei der zeitgesteuerten Simulation wird die Simulationszeit um ein konstantes Zeitinkrement Δt erhöht. Zustandsveränderungen, die innerhalb des letzten Zeitfensters (Epoche) aufgetreten sind, werden anschließend ausgeführt. Finden in mehreren aufeinanderfolgenden Epochen keine Zustandsveränderung statt, so werden die Epochen dennoch nach einander durchgeführt [Mat89 S202 f.].

2.6.3 Computerspiele und Echtzeitsimulation

Die Berechnungen in Computerspielen basieren zu einem großen Teil auf mathematische Modelle. Dabei werden Objekte abhängig von der Zeit und anderen Objekten beeinflusst bzw. verändert. In der Regel werden hier abstrahierte Abbildungen der Realität verwendet. Hinzu kommt die Möglichkeit der Interaktion des Spielers in Echtzeit mit der Simulation. Computerspiele haben demnach in vielen Teilen einen Ablauf, der denen von Simulationen sehr ähnelt. In [Gre09 S9] werden Computerspiele als *„soft real-time interactive agent-based computer simulations“* beschrieben.

Hervorzuheben ist, dass es in einem Computerspiel im Gegensatz zu einer Simulation eine Deadline gibt. Die Deadline ist der Zeitpunkt, an dem ein Simulationsschritt vollendet sein muss. Alle Kalkulationen, die für den nächsten Zeitschritt nötig sind, müssen vor der Deadline durchgeführt sein. In Computerspielen ist die Deadline typischerweise der Zeitpunkt für die Berechnung des nächsten Einzelbildes (Frame) [WikiFrame] [WikiFrame2].

Wird eine Deadline nicht eingehalten, so kann dies das Spielerlebnis trüben (Ruckler, Nicht-Verarbeitung von Nutzereingaben usw.) [Gre09 S10].

Ein Simulationsschritt wird auch als Tick bzw. GameTick bezeichnet. In den meisten Multiplayer Spielen kommen üblicherweise feste Tick-Raten vor. Diese variieren allerdings stark in ihrer Frequenz. In dem First-Person-Shooter Counter-Strike ist eine möglichst hohe Tick-Rate das präferierte Ziel und es wird mit 66-100 oder mehr Ticks pro Sekunde gearbeitet [CounterStrike]. Echtzeit-Strategiespiele arbeiten dagegen mit einer geringeren Tick-Rate (5-50 Ticks pro Sekunde [AgeOfEmpires] [Smith1] [Starcraft2]), da die Berechnung der Spielobjekte mehr Zeit kostet und die Reaktionsgeschwindigkeit keine ganz so große Rolle spielt.

Es werden zudem auch teilweise verschiedene Tick-Raten in einem Spiel eingesetzt, um das Spiel performanceschonender zu betreiben. Hier werden dann nur für Zeitkritische Aspekte hohe Tickraten verwendet, um eine schnelle Reaktion zu garantieren[RuneScape].

3 Anwendungsszenario

In dieser Arbeit wird von einem Spiel ausgegangen, in dem die Aufgabe des Spielers in der Organisation von Produktionsketten liegt. Dabei muss der Spieler sich der Rohstoffförderung, sowie deren Weiterverarbeitung und der Forschung(Erforschung von besseren, im Spiel definierten, Technologien) widmen. Das Verkehrsnetz dient dabei dem Transport von beliebigen Objekten, wie Waren oder Personen. Ein im Spiel existierendes Stahlwerk könnte beispielsweise nur in Betrieb genommen werden, wenn es genügend Arbeitnehmer, Kohle für die Hochöfen und genügend Roheisen zur Verfügung hat. Die Anlieferung dieser benötigten Ressourcen muss über das Verkehrsnetz geschehen. Dasselbe gilt für einen Viehzuchtbetrieb. Er muss an einer Straße angeschlossen sein, um mit den benötigten Ressourcen versorgt zu werden. Als benötigte Ressourcen wären hier verschiedene Getreidesorten, Medikamente, Wasser, Treibstoff o.Ä. denkbar. Auch der Abtransport von Abfällen kommt in Frage. Bei der Holzgewinnung muss der Spieler für die Erweiterung des Verkehrsnetzes sorgen, wenn Wälder gerodet wurden und er für den Erhalt seiner Produktion neue Bereiche der Spielkarte erschließen muss, um an bestehende Wälder zu gelangen.

Abstrahiertes Spielszenario

In dieser Arbeit wird eine stark vereinfachte Produktionskette verwendet. Fahrzeuge können Aufträge annehmen, die in der Spielwelt verteilt sind. Diese Aufträge sind erfolgreich bearbeitet worden, wenn das Fahrzeug die Koordinate des Auftrags erreicht hat. Dabei muss das Fahrzeug den Auftrag über das Verkehrsnetz erreichen. Ist ein Auftrag erledigt, versucht das Fahrzeug, einen neuen Auftrag zu erhalten. Weiterhin ist das Verkehrsnetz in dieser Arbeit auf Straßen beschränkt. Ergänzend könnte es auch Wasserwege, Schienen und Flugwege umfassen.

4 Analyse

Der erste Teil dieses Kapitels befasst sich mit existierenden Ansätzen bei der Verwaltung von Objekten in Spielen, speziell MMOs, und beleuchtet einige wichtige Aspekte in der Entwicklung von (Spiel-) Serversystemen. Es wird versucht, Problemstellen ausfindig zu machen und anschließend im zweiten Teil des Kapitels diese bei der Erstellung der Anforderungen in Betracht zu ziehen. Dabei erfolgt zunächst eine Zusammenfassung der gewonnenen Erkenntnisse, und es wird ein angestrebtes Spielkonzept, basierend auf diesen Erkenntnissen, genauer beschrieben.

Die Anforderungen unterteilen sich in funktionale und nichtfunktionale Anforderungen. Dabei wird das in Kapitel 3 vorgestellte Anwendungsszenario einbezogen. Die Anforderungen beziehen sich auf das Verkehrsnetz, sowie auf einen Spiel-Prototyp, in dem das Verkehrsnetz implementiert wird und welcher vom Verkehrsnetz mit benutzt werden muss. Andere Teilsysteme einer möglichen Serverstruktur werden nicht beachtet. Bei der Beschreibung der Anforderungen wird auf den Prototyp der Spieleumgebung eingegangen, bevor die gleichen Informationen für das Verkehrsnetz aufgestellt werden.

4.1 Analyse von existierenden Spielen und Konzepten

In diesem Abschnitt werden einige Konzepte und Herangehensweisen vorgestellt, die in existierenden Spielen eingesetzt werden oder von Spieleentwicklern eingesetzt wurden. Viele der Informationen stammen aus Artikeln und Blogs, von Game Designern oder Programmierern. Grund hierfür ist die sehr eingeschränkte Literaturauswahl im ‚Multiplayer Game Development'-Bereich. Einige ausgewählte Werke bieten Einblicke in die Erfahrungen von Firmen, die meisten Bücher stellen aber nur Basisinformationen zur Verfügung, die für den speziellen Bereich der MMOs nicht ausreichend detailliert sind. Ausschlaggebend für diesen Zustand ist das noch sehr junge und durchgehend im Wandel befindliche Spielgenre selbst. Einige Detaileinblicke bietet [Ale05], dessen Informationen allerdings auch bereits einige Jahre zurück liegen.

Synchronisierte Engine Architektur

Der bekannt gewordene Artikel *"1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond"* [AgeOfEmpires] über den Multiplayer Designprozess von Age of Empires 1 & 2 beschreibt eine sehr bandbreitenschonende Netzwerkarchitektur. Dabei werden lediglich Nutzer-Eingabeinformationen übertragen und keine Positionsinformationen von Einheiten und Gebäuden. Notwendig für eine solche Architektur sind eine synchrone Ausführung aller Aktionen auf allen Teilnehmern der Multiplayersitzung und ein deterministischer Ablauf der Anwendung (Dieser Vorgang wird im Folgenden als Resimulation bezeichnet [Ale05 S201] [Hal08 S773 ff.]). Demnach müssen auch alle Pseudo-Zufallsgeneratoren auf dem gleichen Startwert arbeiten und die gleichen Zufallswerte ergeben. Die Kommunikation wurde dabei über eine Peer-to-Peer Architektur gelöst.

Ein weiterer bekannt gewordener Artikel über die Architektur von synchronen RTS-Games *"Synchronous RTS Engines and a Tale of Desyncs"* [Smith1] beschäftigt sich hauptsächlich mit dem vom Autor mitentwickelten Supreme Commander [WikiSupCom]. Dabei wird auch auf die Übertragung von Nutzereingaben zu allen Teilnehmern und anschließender Resimulation gesetzt. Ein zweiter Artikel [Smith2] wurde vom Autor veröffentlicht, in dem die Funktionsweise der Synchronisation zwischen den Teilnehmern genauer erläutert wird. Der Autor geht zudem auf einige weitere Spiele ein, die nach dem gleichen Konzept funktionieren wie Supreme Commander. Darunter sind beispielsweise Starcraft, Command & Conquer und Halo.

Floating Point Determinismus

Die Berechnung von exakt der gleichen Simulation auf verschiedenen Hostsystemen erfordert eine besondere Berücksichtigung von Floating-Point-Kalkulationen. In [Fiedler] wird auf diese Problematik eingegangen. Durch unterschiedliche Techniken zur Performancesteigerung kann das Ergebnis einer Floating-Point-Kalkulation sowohl von Hardware zu Hardware, als auch von Compiler zu Compiler unterschiedlich sein. Ein über verschiedene Plattformen einheitlicher Determinismus scheint dem Artikel nach nur über besondere Parameter möglich zu sein. Als Lösung wird von in dem Artikel die Einhaltung des IEEE 754 Standards [WikiIEEE754] auf den Hostsystemen angegeben, welcher die Performanz der Floating-Point-Kalkulation beeinträchtigt. Angesprochen wird zudem der mögliche Verzicht auf Fließkommawerte im für die Simulation essentiellen Bereich, da ganze Zahlen von diesem Problem nicht betroffen sind.

MMO Back-end Architektur

Der Artikel *„GDC 2005 Proceeding: Online Game Architecture: Back-end Strategies"* [Esbensen] von der Game Developer Conference 2005 über die Infrastruktur von MMORPGS gibt einen kleinen Einblick in verschiedene Problemfelder von MMO Serversystemen. Es wird unter anderem angesprochen, dass Everquest [Everquest], ein MMORPG, zwischen 200 und 300 Spielern auf einem Server verwalten kann und die damit verbundenen Serverkosten ein hohes wirtschaftliches Risiko für die Betreiberfirma darstellen. Der Artikel geht zudem auf den Effekt von Lags (Ruckler im Spielfluss, Erhöhte Verzögerung zwischen Aktion und Ausführung) auf das Spielerlebnis der Teilnehmer ein. Diese können einen negativen Eindruck über das Spiel unter den Spielern erzeugen, und dem Image des Spiels und der Firma schaden.

Eve Online

Eve Online [Eve1] ist ein Weltraum MMORPG, in dem der Spieler in die Rolle eines Piloten schlüpft. Dabei spielen alle Spieler zusammen in einer einzigen riesigen Galaxie. Die Lastverteilung wird über die Einteilung in Zonen vorgenommen. Die Galaxie unterteilt sich in tausende Sonnensysteme, von denen jedes ein einzelner Prozess ist. Die Prozesse können auf unterschiedlichen Servern ausgeführt werden und die Last so verteilt werden. Problematisch erweisen sich bei diesem Konzept Sonnensysteme mit einer hohen

Spieleranzahl bzw. hohen Beliebtheit unter Spielern. Die Spielerzahl in einem sehr beliebten Sonnensystem musste sogar, limitiert werden [Eve2]. Eve Online arbeitet mit Stackless Python und den darin integrierten Microthreads [StacklessPython].

Second Life

In [Ale05 S72 ff.] wird die Architektur von Second Life [SecondLife] dargestellt. Auch SecondLife setzt auf die Zerteilung der Welt in einzelne Abschnitte. Dabei wird einem Server jeweils eine Region zugewiesen. Die Kommunikation zwischen den Servern und den Clients wird über UDP gelöst und ist nach Ian Wilkes, VP of Systems Engineering at SecondLife, eine im Nachhinein schlechte Entscheidung [Wilkes]. Grund für die Nutzung von UDP war die Annahme, dass ein Großteil der Objekte in der Spielwelt sich bewegen und häufigere Positionsupdates benötigen würden, was sich nicht bestätigen sollte. Die Simulationseinheiten für die Spielwelt auf den Server laufen mit 45 Ticks pro Sekunde. Kommt es zu einer hohen Belastung des Systems, wird die Rate halbiert, und das Spiel läuft für die Nutzer in Zeitlupe ab. Auf den Einsatz von Multithreading in den Simulationseinheiten wird verzichtet, da diese Technik mehr Probleme mit sich bringen würde, als die Optimierung und Skalierung einer singlethreaded Anwendung [Wilkes]. Aufgrund der besonderen Größe des Spiels, und weil die Welt von den Usern vollständig manipuliert werden kann, werden alle für den Client benötigten Daten an die Teilnehmer vom Server gestreamed.

Siedler 7

In Siedler 7 existiert ein komplexes Bau- und Produktionssystem. Der Transport von Waren zwischen den einzelnen Gebäuden wird von Arbeitern übernommen. Sie müssen Waren aus einem Gebäude abholen und zu einem anderen bringen, bevor der Betrieb anfangen kann zu Arbeiten. Die Arbeiter können sich nur auf vordefinierten Wegen bewegen und laufen durch einander durch, wenn sich zwei Arbeiter begegnen. Zwischen militärischen Einheiten und Arbeitern findet keine Interaktion statt. Der Multiplayer Modus ist auf nur vier Teilnehmer beschränkt [Siedler].

CitiesXL

CitiesXL [CitiesXL] ist eine Wirtschaftssimulation und fällt in die Kategorie der *„city-building games“*. Das von Monte Cristo entwickelte Spiel besaß einen Multiplayer Modus, in dem Spieler ausgewählte Waren wie Strom, Wasser oder Treibstoff unter einander Handeln konnten. Dieser Handel blieb weiterhin erhalten, wenn ein Spieler nicht online war [WikiCitiesXL]. Die Städte wurden dabei auf einem Serversystem gespeichert und vollständig an die Spieler übertragen. Die Manipulation von Spielinhalten wurde nicht zur Laufzeit vom Serversystem überprüft. Dies hat eine Manipulation des Onlinehandels zwischen den Städten erlaubt. Ein Spieler konnte zum Beispiel eine Stadt mit sehr vielen Kraftwerken erschaffen, die eine negative Geldbilanz aufwies, und das Spiel verlassen. Im zweiten Schritt hat er eine neue Stadt erschaffen und den Strom seiner ersten Stadt für einen minimalen Preis eingekauft.

Das Verkehrsnetz in CitiesXL transportiert keine Waren. Die Fahrzeuge werden über statistische Werte simuliert.

SimCity (GlassBox Engine)

Das Spiel Sim City 5 [SimCity5] befindet sich aktuell in der Entwicklung und wird in mehreren Videos von den Entwicklern genauer erläutert [SimCityGlassBox]. Das Spiel setzt stark auf Agenten, die bestimmte Aufgaben übernehmen. Beispielsweise generiert ein Wasserturm in regelmäßigen Abständen Simulationsagenten, die sich über das Straßensystem bewegen und Wasser an die Häuser liefern [SimCity5YT]. Auch Personen agieren als Agenten und müssen ihren Arbeitsplatz über das Verkehrsnetz im Spiel erreichen. Die Größe der Städte ist dabei stark reduziert worden im Vergleich zum Vorgänger Sim City 4 [WikiSimCity]. Der Handel im Multiplayer Modus soll auf die gleiche Weise funktionieren wie in CitiesXL. Eine genauere Beschreibung, gibt es bisher leider noch nicht. Eine Aussage über den Schutz gegen Manipulation im Multiplayer Modus wurde noch nicht gemacht [SimCity5YT].

OpenTTD

OpenTTD [OpenTTD] ist eine Open-Source Wirtschaftssimulation, die dem originalspiel Transport Tycoon [TransportTycoon] nachempfunden wurde. Hauptaufgabe des Spielers ist der Transport von Waren zwischen verschiedenen Produktionsgebäuden, die nicht unter Spielerkontrolle stehen. Auch dieses Spiel setzt auf Determinismus und Resimulation. OpenTTD unterstützt bis zu 255 Spieler. Der Multiplayer Modus ist dabei so strukturiert, dass ein Serversystem die Interaktionen der Spieler an alle anderen Spieler verteilt. Haben alle Spieler die gleichen Informationen erhalten, wird die Simulation um eine Zeiteinheit erhöht. Verbindet sich ein Spieler neu mit dem Spiel, wird pausiert und der aktuelle GameState an den Spieler versendet. Anschließend läuft die Simulation synchronisiert weiter.

Minecraft

Das Open-World-Spiel Minecraft [Minecraft] bietet dem Spieler im Singleplayer-Modus als auch im Multiplayer Modus eine dynamisch generierte Spielwelt, die vollständig manipulierbar ist. Im Multiplayer Modus übernimmt der Server alle spielentscheidenden Berechnungen und teilt Veränderungen an der Welt den Spielern mit. Die Simulation der Umgebung wird nur in der direkten Umgebung von Spielern ausgeführt. Gebiete, die weiter von Spielern entfernt sind, werden pausiert und entladen. Das Spiel läuft mit 20 Ticks pro Sekunde und wurde in Java entwickelt. Minecraft hat in den letzten Jahren einen enormen Bekanntheitsgrad erlangt [WikiMinecraft] und wurde besonders als Multiplayer-Spiel über die Mehrspielermodifikation Bukkit [Bukkit] populär, die es erlaubt, beliebige Modifikationen an dem Spiel vorzunehmen. Ein Minecraft-Server kann bis zu 100 Spieler verwalten (Erfahrungswert des Autors). Durch von Spielern vorgenommene Modifikationen, die den Spielumfang auf verschiedene Arten einschränken, ist es möglich bis zu einigen hundert Spielern auf einem Server zu verwalten (Erfahrungswert des Autors).

Browsergames

In vielen Browsergames brauchen Aktionen des Spielers eine gewisse Zeit, bis diese vollendet wurden. Beispielhaft können hier Die Stämme, Travian und Farmville [Stämme] [Travian] [Farmville] genannt werden. In Browsergames werden die Änderungen an der eigenen Spielwelt nur vorgenommen, wenn der Nutzer aktiv (in das Spiel eingeloggt) ist. Verlässt der Spieler das Spiel, wird die Welt pausiert und die Veränderungen an der Welt erst vorgenommen, wenn der Nutzer sich wieder anmeldet. Dazu wird mit Zeitstempeln gearbeitet, um den Fortschritt berechnen zu können, wenn der Nutzer sich erneut anmeldet. Ist das Spiel stark auf Kampfhandlungen ausgelegt, wie beispielsweise in Die Stämme, so werden zusätzlich zukünftige Zeitpunkte gespeichert, an denen z.B. ein Angriffstrupp bei einer Stadt ankommt. Ist der Zeitpunkt erreicht, führt das System diese Aktion aus, auch wenn beide, von der Kampfhandlung betroffenen Spieler, nicht in diesem Moment aktiv sind. Es war nicht möglich eine Quelle für die Funktionsweise von den genannten Spielen zu finden. In [Klausa S27] wird der vorgestellte Ansatz verwendet.

Besonderheiten von Multiplayer Echtzeit-Strategiespielen

Bei den Multiplayer Echtzeit-Strategiespielen läuft das Spiel in einem festen zeitlichen Rahmen ab, und auch die Anzahl der Spieler bleibt meist stark beschränkt. Mehr als 8 Spieler in einem Spiel ist eine Seltenheit. Der Sprung in ein neues Genre, wie es bei den RPGs zu den MMORPGs gelungen ist, steht noch aus.

Ein entscheidender Grund hierfür ist die Komplexität der Spieleumgebung [Smith2]. Der Verwaltungsaufwand eines Strategiespiels ist durch die Vielzahl an Objekten in der Spielwelt erheblich höher als bei Rollenspielen. Die Objektzahl ist dort teilweise auch hoch, die Objekte sind aber für das Spielgeschehen nicht weiter wichtig, sobald sie sich außerhalb des Sichtradius des Spielers befinden. In einem Strategiespiel haben üblicherweise sehr viele Objekte Einfluss auf das Spielgeschehen, auch wenn sie außerhalb des Sichtradius des Spielers sind. Hinzu kommt das Problem der Umweltmanipulation. Ist der Spieler in der Lage, die Umgebung bleibend zu verändern, steigt zum einen der Verwaltungsaufwand, und zum anderen könnte das Spieleerlebnis für nachfolgende Spieler grundlegend verändert werden. Ein Gebiet, welches zu Beginn des Spiels sehr anfängerfreundlich war, könnte durch die Manipulation der Umwelt anfängerunfreundlich werden. Ein Effekt, der ein hohes Risiko für den Erfolg des Spiels und der Entwicklungsfirma darstellen kann [Hal08 S54 f.]. In MMORPGs wird typischerweise eine statische Welt verwendet, die nur kurzzeitigen Änderungen durch Spieler unterliegt. Jede Änderung an der Welt (Abgebaute Ressourcen) wird nach wenigen Minuten zurückgesetzt. Der nächste Spieler kann die Änderung erneut vornehmen. Dieses Konzept ist prinzipiell auch auf Strategiespiele übertragbar, beispielhaft kann das in der Beta-Phase befindliche Spiel End of Nations genannt werden [EndOfNations]. Dieses ist ein Strategiespiel, verhält sich aber in vielen Teilen wie ein MMORPG. Spielkarten unterliegen nur temporären Änderungen, der Spieler hat ähnlich wie in Rollenspielen die Möglichkeit, seine Charaktere/Fahrzeuge weiter zu entwickeln. Verlässt ein Spieler das Spielgeschehen, wird nur der Zustand seiner Erfahrungen und Einheiten gespeichert. Die Informationen der

Spieleumgebung gehen verloren und werden bei der nächsten Spielsession neu erstellt. Die Anzahl an Strukturen ist zudem stark limitiert und die Karte selbst kann nicht verändert werden [EndOfNations2].

Verkehrsnetz Simulationen
Neben den verschiedenen Techniken, die in Spielen eingesetzt werden, wurden noch Konzepte analysiert, die in Verkehrssimulationen verwendet werden. Neben der in Spielen verwendeten Diskreten Event Simulation existiert noch die Umsetzung über zelluläre Automaten [Nag99]. Diese eignen sich allerdings auf Grund der Neuberechnung aller Fahrzeugpositionen in jedem Simulationsschritt [Hat07] nicht.

Zusammenfassung
In diesem Abschnitt wurden mehrere Texte, Artikel und Spiele analysiert. Viele der Informationen sind Erfahrungen von Spieleentwicklern und beziehen sich auf sogenannte Bottlenecks (also Engpässe), die sich je nach Spiel- oder Softwaretyp stark unterscheiden. Als nächstes werden die wichtigsten Erkenntnisse zusammengefasst. Es folgt eine genauere Beschreibung des in dieser Arbeit angestrebten Spielkonzepts. Das Ende des Kapitels bildet die Erstellung der Anforderungen an den Spielprototyp und das Verkehrsnetz.

4.2 Ergebnis der Analyse

In diesem Teil des Kapitels werden die wichtigsten Erkenntnisse aus der Analyse zusammengefasst und im Anschluss die Anforderungen für den Spielprototyp und das Verkehrsnetz definiert.

Das hohe Objektaufkommen, speziell das Einheitenaufkommen, stellt in Strategiespielen eine besondere Herausforderung dar.
Die Verwaltung und Berechnung, also die Aktualisierung von Positionsinformationen und die Prüfung von Kollisionen, benötigt viel Prozessorzeit. Der Verwaltungsaufwand in Multiplayer Spielen ist dabei entsprechend erhöht, da ein zentraler Server die Verwaltung von allen Objekten übernehmen muss. Vergleicht man aktuelle Spieletitel der Genres mit einander (RTS im Vergleich zu (MMO)FPS, (MMO)RPG), macht sich dies durch einen Unterschied in der Maximalen Spielerzahl bemerkbar, welche bei Strategiespielen entsprechend niedriger sind. Das Verwaltungsproblem wird auf verschiedene Arten gelöst oder umgangen. Browsergames verzichten größtenteils auf direkte Interaktion mit Spielobjekten, zudem brauchen Aktionen wesentlich mehr Zeit als in herkömmlichen Computerspielen, was eine enorme Entlastung der Serversysteme darstellt. Das Spiel SimCity5 wird, den aktuellen Informationen nach [SimCityGlassBox] [SimCity5YT], im Singleplayer-Modus ablaufen, der Multiplayer Modus wird nur über den Handel von Ressourcen zwischen Städten realisiert. Die Komplexität der Städte ist enorm, sie werden aber stark verkleinert, um sie lauffähig zu halten. In anderen Spielen wird die Zahl der Teilnehmer auf eine niedrige Zahl begrenzt oder die Anzahl an Objekten eingeschränkt (Anno, Siedler 7, End of Nations, Age of Empires).

Die Verwaltung von großen zusammenhängenden Welten ist schwierig.
Die Welten werden häufig in Regionen eingeteilt und einem Server oder Serverprozess zugewiesen. Dabei wird größtenteils auf Multithreading innerhalb eines Prozesses verzichtet. Hauptargument dafür ist die einfachere Skalierung der Hostsysteme und die Vermeidung von erhöhter Codekomplexität durch Multithreading. Ein oft verwendetes Konzept ist auch die Limitierung der Teilnehmer innerhalb einer Welt. Neue Spieler werden dann gezwungen, auf einer zweiten Welt anzufangen, die unabhängig von der ersten ist.

Dauerhafte Veränderungen an der Spielwelt sind problematisch.
Es existieren nur sehr wenige Multiplayer Spiele, in denen die Welt dauerhaft manipulierbar ist. Die Erhaltung von Spielspaß und Spielerfahrung stellt hier ein Problem dar. Eine sich verändernde Welt kann leicht zu veränderten Spielerfahrungen unter den Spielern führen. Ein zu Anfang gut geplanter und durchdachter Bereich des Spiels, der extra für neue Spieler gedacht war, könnte durch die Manipulation von Spielern mit der Zeit nicht mehr anfängerfreundlich sein.

Die verzögerte oder langsame Reaktion des Spiels sollte vermieden werden.
Der Spielspaß wird stark von dem reibungslosen Ablauf des Spiels beeinflusst. Auch eine langsame Reaktion auf Interaktionen kann hier einen schlechten Einfluss haben. Andererseits erlaubt eine höhere Verzögerung den Serversystemen mehr Zeit zur Berechnung. Es muss demnach eine gute Balance zwischen Spielfluss und Performanceaufwand gefunden werden.

Das Ereignisaufkommen sollte möglichst gering gehalten werden.
In den Spielen wird versucht, die Anzahl der Ereignisse gering zu halten. Sehr gut sichtbar ist dies bei Browsergames, die lange Ruhephasen zwischen Aktionen haben. Auch in Simulationen wie SimCity 4 werden statistische Werte generiert [SCTraffic]. In Multiplayer Spielen ist dies besonders wichtig, da ein Ereignis oft die Versendung dieser Informationen an den Client nach sich zieht. Werden in einem Spiel viele Events generiert kann dies bei einer hohen Anzahl an Spielern, die diese Information benötigen, schnell zu Netzwerkproblemen führen.

Spiele mit direkter Bewegungssteuerung haben mit vielen Usernachrichten zu kämpfen.
Spiele, in denen es dem Benutzer möglich ist, einen Charakter direkt mit Tasteneingaben zu steuern, benötigen eine hohe Paketrate, um Charakterbewegungen flüssig darzustellen. In Spielen, welche die Bewegung von Objekten nur indirekt über Aktionen erlauben (Anno 1404, Supreme Commander), tritt dieses Problem nicht auf.

Echtzeit-Strategiespiele haben häufig mit Determinismus und hoher CPU-Last zu kämpfen.
Sie erfordern in der Regel eine geringere Reaktionsgeschwindigkeit als First-Person-Shooter oder Rollenspiele, dafür haben sie ein sehr hohes Informationsaufkommen, welches eine riesige Masse an Netzwerkpaketen produzieren würde. Aus diesem Grund wird in Echtzeit Strategiespielen auf synchronisierte Resimulation gesetzt. Hierbei fallen nur Datenpakete an,

wenn der User mit dem Spiel interagiert und damit den Verlauf der Simulation verändert. Andernfalls kann jeder Teilnehmer des Spiels den Fortschritt durch den Determinismus selber errechnen.

Die Transportnetze in Spielen sind meist einfach gehalten.
Der Transport von konkreten Waren wird nur selten in Strategiespielen implementiert. Oft werden Ressourcen an einem Standort gefördert und sofort in ein globales Lager übertragen, auf das von überall aus zugegriffen werden kann. In Anno 1404 werden Rohwaren direkt von anliegenden Gebäuden abgeholt oder über Straßen in Lagerstätten transportiert. Siedler 7 setzt für den Bau von Gebäuden und für die Produktion von Waren voraus, dass die nötigen Materialien zuvor von Einheiten an die Gebäude oder die Baustellen geliefert wurden. Beide Spiele ignorieren bei diesem Transport die Kollision zwischen Transporteinheiten. Eine Ausnahme bildet OpenTTD. Dieses Spiel besitzt ein sehr komplexes Transportsystem. Das Spielkonzept basiert ausschließlich auf dem Transport von Waren.

Die Bewegung von Objekten wird in jedem Zeitschritt der Echtzeitspiele neu kalkuliert.
In allen untersuchten Spielen, abgesehen von Browsergames, wird der Zustand von Objekten periodisch überprüft. Das bedeutet, sie werden in jedem Zeitschritt oder in vordefinierten Zeitschritten aufgerufen, und es findet eine Analyse ihres aktuellen Zustandes statt. Je nach Ergebnis dieser Analyse wird eine Anpassung des Zustandes vorgenommen. Bei der Analyse werden beispielsweise Kollisionsprüfungen vorgenommen, nach angreifbaren Einheiten in der Umgebung gesucht, das Erreichen eines temporären Ziels untersucht oder ein Umgebungseffekt auf die Einheit angewendet (Sonne auf eine Blume führt zu Wachstum der Pflanze).

4.3 Angestrebtes Spielkonzept

Im Folgenden wird das angestrebte Spielkonzept beschrieben. Dafür werden die Analyseergebnisse verwendet. Bemerkt sei hier, dass ein Ergebnis der Analyse nicht bedeutet, dass dieses automatisch übernommen wird. Das vorgestellte Spielkonzept ist ein Kompromiss der Ergebnisse in Verbindung mit dem Vorhaben, weiterhin so viele Freiheiten wie möglich in das Spiel zu integrieren.

Die Spielwelt ist größtenteils manipulierbar
Der Spieler soll in dem Spielkonzept ähnlich wie in OpenTTD oder Minecraft die Möglichkeit haben, die Umgebung mit zu gestalten und zu bearbeiten. Dabei könnte zum Beispiel ein Tagebau modelliert werden, in welchem der Spieler die Aufgabe hat, den Tagebau zu verwalten und für einen effizienten Warentransport zu sorgen.

Der Spieler hat eine hohe Gestaltungsfreiheit seiner Umgebung
Dem Spieler stehen über die Spielmechanik genügend Methoden offen, um seine Umgebung nach seinen Vorstellungen zu gestalten. Es gibt keine vorgeschriebenen Orte, an denen z.B. ein Gebäude gebaut werden muss. Bei dem genannten Tagebau kann der Spieler dann beispielsweise auch Gebiete abbauen, die gar keine für ihn wertvollen Rohstoffe enthalten.

Die Rohstoffförderung und –verarbeitung ist ein wichtiger Teil des Spiels
Die Organisation von Warenketten und die damit verbundene Rohstoffförderung und -verarbeitung bestimmen einen großen Teil des Spiels. Die daraus gewonnenen Ressourcen können vom Spieler zur Erforschung von Technologien oder dem Ausbau seines ‚Imperiums' verwendet werden. Der Tagebau würde beispielsweise zur Kohleförderung dienen. Die geförderte Kohle kann in einem Kraftwerk verwendet, verkauft oder gegen andere Waren getauscht werden.

Der Transport von Gütern ist ein essentieller Teil des Spielkonzepts
Eine wichtige Aufgabe des Spielers ist die Organisation von Transporten. Der Spieler muss eine effiziente Struktur innerhalb seiner Warenketten und Transportnetze besitzen, um eine optimale Ausschöpfung seiner Möglichkeiten sicherzustellen. Ihm steht zum Beispiel die Möglichkeit offen, die Verkehrsanbindung des Tagebaus so zu gestalten, dass der Verkehrsfluss optimal ausgeschöpft wird und keine Fahrzeuge auf ihrem Weg von sonstigem Verkehr behindert werden.

Der Spieler muss den Verkehr selber optimieren, um effizient Ressourcen zu fördern und zu verwalten
Die Verwaltung des Logistiksystems und der Transportmittel muss vom Spieler optimiert werden. Zwar wird eine grundlegende Wegfindung über das Spiel sichergestellt, hat der Spieler aber eine ineffiziente Transportstruktur, wird keine Optimierung vom Spiel selber vorgenommen. Der Tagebau könnte zum Beispiel durch ungünstig strukturierte Fahrwege oder von Transportfahrzeugen anderer Anlagen negativ beeinflusst werden.

Jeder Spieler hat seine eigene Spielwelt
Die Teilnehmer des MMORTSG interagieren jeweils mit einer Spielwelt, die speziell für sie erstellt wurde. Die Spieler agieren allerdings nicht autark. Die direkte Interaktion vieler Spieler in derselben Spielwelt ist zwar für das Spieleerlebnis wichtig, wird in dieser Arbeit jedoch nicht behandelt. Wie bereits zuvor erwähnt wird in dieser Arbeit nicht auf die Vernetzung der Software eingegangen.

Das Spiel verlangt keine hohe Reaktionsgeschwindigkeit
In vielen Spielen ist eine hohe Reaktionsgeschwindigkeit ausschlaggebend für den Erfolg. Das hier behandelte Spiel verlangt allerdings keine hohen Reaktionsgeschwindigkeiten und ist vergleichbar mit den Anno-Titeln, SimCity oder Siedler.

Das Spiel lässt sich unterbrechen und später fortsetzen
Verlässt ein Spieler das Spiel, so kann es serverseitig pausiert und entladen werden. Der Spieler kann zu einem späteren Zeitpunkt zurückkehren und das zuvor unterbrochene Spiel wieder aufnehmen. Seine Welt geht bei der Spielunterbrechung nicht verloren.

Die direkte Steuerung von Objekten ist nicht möglich
Der Spieler kann den Objekten nur Befehle geben. Es ist ihm nicht möglich, wie in First-Person-Shootern, ein Objekt (Spielfigur) direkt zu beeinflussen/steuern.

4.4 Anforderungen

Die zuvor genannten Erläuterungen zum Spielkonzept werden verwendet, um Anforderungen sowohl für die Spieleumgebung, als auch für das darin zu integrierende Verkehrsnetz, zu definieren. Im Folgenden werden zunächst Anforderungen an die Spielumgebung bzw. den Spiel-Prototyp aufgestellt. Diese beinhalten das Zur-Verfügung-Stellen von grundlegenden Funktionalitäten für die Realisierung des Verkehrsnetzes. Die Anforderungen orientieren sich, wie in der Zielsetzung bereits vermerkt, auch an der Erweiterbarkeit der Spielumgebung. Sie soll eine Entwicklung des Konzepts und des Verkehrsnetzes über diese Arbeit hinaus begünstigen. Im Anschluss werden dann die Anforderungen an das Verkehrsnetz aufgelistet.

Die Definition von Anwendungsfällen wird nicht vorgenommen, da es sich in dieser Arbeit um ein Forschungsprojekt handelt und nicht um die Entwicklung einer nutzbaren Software für Anwender. Im Verlauf der Arbeit wird eine RenderEngine für die Visualisierung des Spiels entwickelt. Diese erlaubt die Erzeugung eines Bildes aus den Rohdaten des Programms [Gre09 S399 ff.] [Rendering]. Sie wird verwendet, um das Verkehrsnetz auch auf Probleme testen zu können, die durch Komponententests nur schwer zu finden sind [Bat04 S176 ff.]. Es werden keine besonderen Anforderungen an die RenderEngine definiert. Der Testvorgang wird in Kapitel 7 beschrieben.

4.4.1 Spiel Prototyp

Für die Implementierung eines Verkehrsnetzes muss die Spieleumgebung grundlegende Eigenschaften beinhalten und die im Spielkonzept genannten und auf das Verkehrsnetz bezogenen Punkte unterstützen.

Funktionale Anforderungen

- Es muss eine Umgebungswelt geben, in der das Spiel stattfindet
- Die Umgebung sollte manipulierbar sein
- Es muss eine Möglichkeit geben, Einheiten und Gebäude innerhalb der Umgebung zu erstellen
- Es muss Aufträge geben, die von Einheiten angenommen und abgearbeitet werden können
- Die Umgebung muss die Möglichkeit liefern, Straßen zu platzieren.
- Fahrzeuge müssen ausstehende Aufträge annehmen können

Nichtfunktionale Anforderungen

Im Vordergrund steht bei den nichtfunktionalen Anforderungen das Verbrauchsverhalten nach der ISO/IEC 9162 [ISO]. Zudem wird versucht weitere Qualitätsmerkmale zu beachten, speziell die Modifizierbarkeit und Testbarkeit. Da es sich bei dieser Arbeit aber um ein Forschungsprojekt handelt, wird wenig Zeit in Bereiche wie Zuverlässigkeit, Benutzbarkeit oder Übertragbarkeit investiert. Die für das Projekt als relevant betrachteten nichtfunktionalen Anforderungen sind:

- **Das Spiel sollte einige Tausend Einheiten und Aufträge verwalten können**
  Das Ziel ist mindestens 3200 Einheiten verwalten zu können (siehe Abschnitt *4.4.2*).
- **Es wird großen Wert auf die Effizienz, speziell das Verbrauchsverhalten, gelegt**
  Das Verbrauchsverhalten unterteilt sich in die benötigten Rechenschritte (Zeitverhalten) und in den benötigten Speicherbedarf (Platzkomplexität). Das Zeitverhalten hat in dieser Arbeit eine weitaus höhere Bedeutung als die Platzkomplexität. Existieren keine Aufträge für Einheiten in der Spielwelt, soll die Prozessorlast gegen null gehen, auch wenn sich mehrere Tausend Einheiten in der Spielwelt befinden.
- **Einzelne Komponenten sind individuell testbar**
- **Das Spiel muss deterministisch sein**
  Dies ist für ein Multiplayer Spiel dieser Art essentiell. Die Gleichheit der Simulation auf verschiedenen Hostsystemen wird in dieser Arbeit nicht berücksichtigt, da die Netzwerkstruktur nicht implementiert wird.

4.4.2 Verkehrsnetz Prototyp

Im Folgenden werden die an das Verkehrsnetz gestellten Anforderungen identifiziert. Auch das Verkehrsnetz wird, wie im Anwendungsszenario bereits angesprochen, in einer prototypischen Komplexität ausgearbeitet.

Funktionale Anforderungen

- Das Verkehrsnetz ermöglicht Fahrzeugen von einem Startpunkt zu einem Endpunkt im Netz zu gelangen.
- Das Verkehrsnetz ist in der Lage, eine Route von einem Start- zu einem Zielpunkt zu berechnen, wenn diese existiert.
- Das Verkehrsnetz ist zur Laufzeit erweiterbar
- Das Verkehrsnetz garantiert, dass Fahrzeuge im Straßennetz nicht kollidieren
- Fahrzeuge im Verkehrsnetz halten sich an vereinfachte Verkehrsregeln (Rechtsfahrgebot, Rechts vor Links)

Nicht-Funktionale Anforderungen

- **Das Verkehrsnetz sollte eine sehr hohe Anzahl an Fahrzeugen verwalten können**
  In den Anforderungen der Spielumgebung wurde bereits erwähnt, dass die Verwaltung von einigen Tausend Einheiten erwünscht ist. Dies ist auch das Ziel für das Verkehrsnetz. Es sollte in der Lage sein einige Hundert, besser einige Tausend, aktive Fahrzeuge im Verkehrsnetz zu verwalten.
- **Das Verkehrsnetz sollte eine gute Effizienz, speziell ein gutes Verbrauchsverhalten, haben**
  Es ist schwierig Vergleichswerte zu finden, die eine Einschätzung der Effizienz des Verkehrsnetzes erlauben. Daher wurde ein grober Vergleichswert auf folgende Weise ermittelt:
  Das in Anno 1404 verwendete Verkehrsnetz ist dem in dieser Arbeit zu entwickelnden am ähnlichsten. Daher wurde ein möglichst komplexer Spielstand dieses Spiels analysiert, der in einem Video zu sehen ist [AnnoYT]. Anhand des Videos wurde die Anzahl an Produktionsgebäuden in der Welt errechnet (ca. 1600). Es wurde davon ausgegangen, dass zu jedem Betrieb zwei Fahrzeuge gehören (Bei einer Holzfällerhütte zum Beispiel der Holzfäller selber, sowie ein Marktkarren, der die Waren abholt). Daraus ergibt sich eine Anzahl von ca. 3200 Fahrzeugen. In der Tablelle [AnnoStats] ist eine Übersicht über die Produktionsgeschwindigkeiten aller Betriebe zu finden (die Werte wurden stichprobenartig geprüft und es wurden keine Fehler gefunden). Der Durchschnittswert liegt bei ca. 30 Sekunden. Das Entspricht 106 Einheitenaufträgen pro Sekunde, wenn die produzierte Ware von jedem Betrieb sofort nach Erzeugung abgeholt wird und nicht mehrere produzierte Einheiten auf einmal abgeholt werden. Das grobe Ziel dieser Arbeit ist daher mindestens 100 Einheitenaufträge pro Sekunde abarbeiten zu können.
- **Das Verkehrsnetz sollte eigenständig testbar sein**
- **Das Verkehrsnetz sollte ‚lebendig' auf den Betrachter wirken**
  Er sollte in der Lage sein, einzelne Fahrzeuge zu erkennen, zu verfolgen und zu analysieren, um gegebenenfalls das Verkehrsnetz auf die Aufgaben der Fahrzeuge optimieren zu können.

4.4.3 Weitere Anforderungen

In diesem Abschnitt werden weitere Anforderungen an das System genannt, die nicht direkt notwendig für die Implementierung sind, die jedoch Beachtung finden sollten, da sie entscheidend für die weitere Entwicklung sind.

Sicherheit
Die Möglichkeiten zu Cheaten sollten so stark wie möglich eingeschränkt werden.

Fehlertoleranz
Der Verlust von Spielobjekten bei einem Clientseitigen Absturz sollte gering sein.

Verbrauchsverhalten
Die Laufzeitkosten der Server-Struktur im Betrieb müssen wirtschaftlich bleiben. Ein einzelner Server sollte in der Lage sein einige hundert Spieler zu verwalten.

5 Konzeption

Dieses Kapitel befasst sich mit der Konzeption der Spielumgebung des Verkehrsnetzes. Als Grundlagen für beide Teile dienen die in der Analyse aufgestellten Anforderungen. Es wird zunächst die Spielumgebung dargestellt und diese als Teil einer komplexeren Serverstruktur erläutert. Der Grund für die Erwähnung der Serverstruktur ist die Teilweise darauf beruhende Konzeption der Spielwelt. Die im Verlauf angesprochene Einteilung des BlockEnvironments in kleinere Regionen und Container wurde beispielsweise gewählt um Daten besser an Spieler streamen zu können und nicht zu viele Informationen über die Umgebung an den Spieler zu liefern. Eine genauere Ausarbeitung und Implementierung der Serverstruktur wird in dieser Arbeit nicht erfolgen. Nachdem der Aufbau und die Funktionsweise der Spieleumgebung und der Unterkomponenten erklärt wurden, wird das Verkehrsnetz behandelt. Dieses soll in das System integriert werden. Anschließend folgt eine detaillierte Erläuterung der Funktionsweise des Verkehrsnetzes. Es wird insbesondere auf die Besonderheiten der Umsetzung eingegangen. Eine Zusammenfassung und Begründung von Designentscheidungen erfolgt am Ende des Kapitels.

5.1 Architektur der Serverstruktur

Die Serverstruktur für MMOs kann sehr komplex werden, da nicht nur die Verwaltung einer Spielwelt benötigt wird, sondern auch weitere Services erstellt werden müssen. Zu diesen gehören beispielsweise Authentifikationsserver, Abrechnungskomponenten oder Nutzersupport-Systeme [Hal08 S44]. Alle Komponenten, die nicht explizit für die Entwicklung des Verkehrsnetzes benötigt werden, entfallen in dieser Arbeit. Weiterhin werden Komponenten, wie eine visuelle grafische Repräsentation des Spiels, durch simplere Mockups (Attrappe) ersetzt. Bei der Architektur wird darauf geachtet, dass das System nachträglich leicht weitere Komponenten und Funktionalitäten integrieren kann. Dieses Konzept wird im weiteren Verlauf genauer erläutert und auch für das Verkehrsnetz verwendet.

5.2 Architektur des Prototyps

In dieser Arbeit wird nur die Simulationseinheit um eine Spielwelt zu verwalten und zu bearbeiten entwickelt. Eine grobe Einteilung des Systems ist in ***Abbildung 1*** zu sehen. Die GameWorld ist die Fassade der Simulationseinheit. Sie kapselt Komponenten, die jeweils unterschiedliche Aufgaben übernehmen. Der Prototyp verfügt zudem über einige Komponenten/Mockups, um das Verkehrsnetzt besser testbar zu machen. Dazu gehören die bereits erwähnte RenderEngine (EnvironmentRenderer), welche eine visuelle Repräsentation der Spielwelt erlaubt, ein Game-Mockup, der rudimentäre Nutzereingaben erlaubt und ein ContentManager (CubedElements), welcher Texturen, HLSL Dateien (High Level Shader Language) und Modelle für die visuelle Darstellung verwaltet. Die Unterkomponenten der GameWorld werden durch Events (Observer Designpattern [Gam94]

[Gre09 S307 ff.] [Ale05 S201 f.] [Ada06 327 f.]) über Spieländerungen informiert. Die einzelnen Komponenten werden im Folgenden beschrieben.

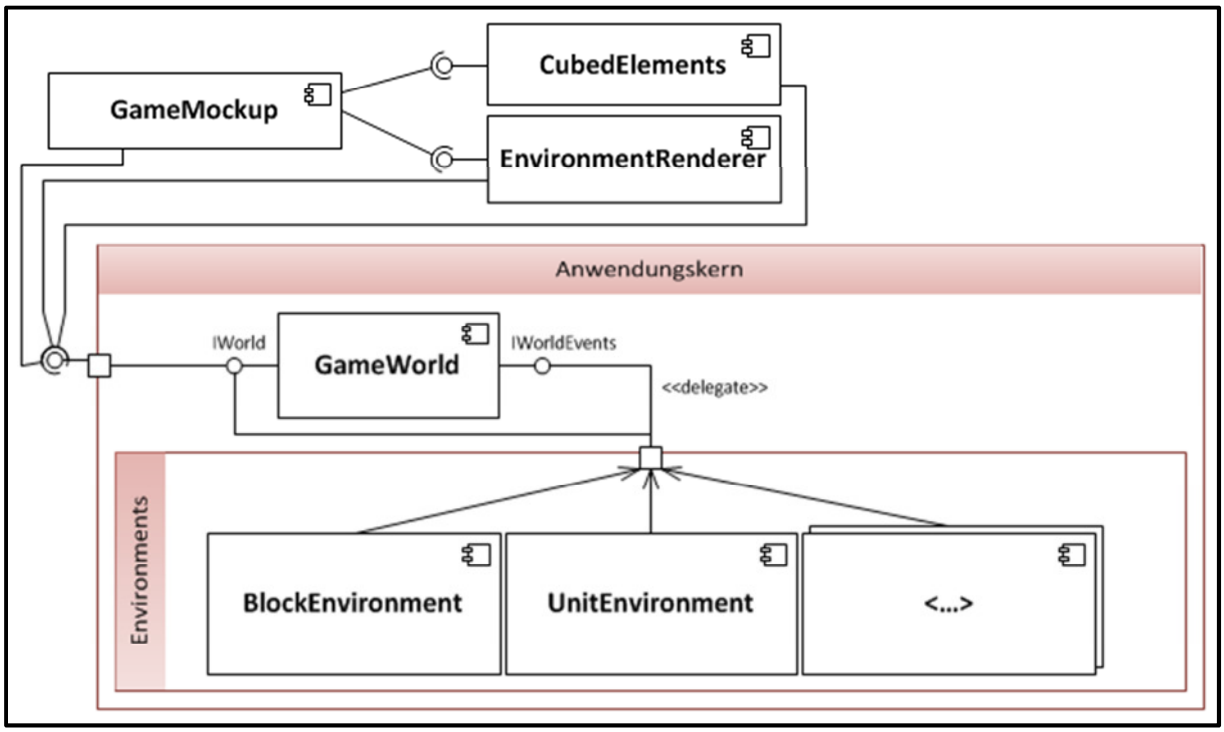

Abbildung 1: Architektur des Prototyps, Kompositionsstrukturdiagramm

5.2.1 GameWorld

Die GameWorld ist die Fassade des Anwendungskerns bzw. des Spielprototyps. Sie kapselt die Unterkomponenten UnitEnvironment, BlockEnvironment, BuildingEnvironment und JobEnvironment. Soll an der Spielwelt eine Änderung vorgenommen werden, darf diese nur über die GameWorld-Fassade stattfinden. Die Unterkomponenten dürfen untereinander keine Änderungen vornehmen. Beispielsweise dürfte das BuildingEnvironment keine Methode des BlockEnvironments zur Entfernung eines Objekts aufrufen. Dieser Aufruf muss über die GameWorld-Fassade stattfinden. Das Verbot ist eine Konvention und muss vom Entwickler eingehalten werden. Ein direkter Zugriff wird nicht explizit verhindert. Die GameWorld stellt alle Methoden zur Verfügung, die von außen aufrufbar sein sollen. Das Testen einzelner Komponenten ist demnach möglich, sobald es eine Implementierung/einen Mockup der GameWorld gibt, da keine weiteren Abhängigkeiten in diesem Stadium der Arbeit in einer Unterkomponente existieren.

5.2.2 Environments

Die Nutzung einer Persistenz-Schicht [Persistenz] zur Speicherung von Änderungen innerhalb einer Spielwelt ist aus performancegründen in Spielen nicht effektiv [Wilkes]. Fängt ein Spieler an, seine Spielwelt zu laden und zu manipulieren, wird diese aus der Persistenz-Schicht in den Arbeitsspeicher/Datenträger einer GameServer-Instanz geladen. Alle Manipulationen der Spielwelt finden auf der GameServer-Instanz statt. Die Persistenz wird

nur erneuert, wenn der Spieler sein Spiel beendet. Aus diesem Grund wird ein performanter Persistenz-Ersatz benötigt, der alle Spielinhalte während der Spielsession aufzeichnet und verwaltet. Diese Aufgabe wird von den Environments (Umgebung, Umwelt) übernommen. Die Environments kapseln verschiedene Aufgabenfelder des Spiels. Das UnitEnvironment ist beispielsweise für die Verwaltung und Validierung aller Einheiten zuständig. In diesem Prototyp gibt es zudem noch das BlockEnvironment, das BuildingEnvironment, sowie das JobEnvironment. Jedes Environment kapselt jeweils die Zuständigkeiten eines Teilbereichs. Die Technik des Persistierens [Gre09 S714] der Spielwelt wird in dieser Arbeit nicht behandelt.

Die Kommunikation unter den verschiedenen Environments findet über ein in Spielen übliches Event-System statt [Gre09 S773]. Dieses orientiert sich an dem Observer-Pattern, welches in [Gam94] beschrieben wird. Aller spielrelevanten Mechanismen und Informationen müssen deterministisch implementiert werden, um die Resimulation auf den Clientmaschinen zu erlauben. Aus diesem Grund darf keine Abhängigkeit zur lokalen Systemzeit implementiert werden und eine Einhaltung des erwähnten Floating-Point Determinismus in Abschnitt ***4.1*** gewährleistet sein. Nur so kann auf den unterschiedlichen Hostsystemen eine Resimulation erfolgen. In dieser Arbeit wird auf Grund der Komplexität des Problems nicht auf die Netzwerksynchronisation eingegangen. Der deterministische Ablauf der Spielsimulation auf einem System, ist dennoch eine notwendige Bedingung in dieser Arbeit, um den fehlerfreien reproduzierbaren Ablauf von automatisierten Tests [Ale05 S173 ff.] zu garantieren.

Im Folgenden werden die Aufgabenfelder der einzelnen Environments erläutert.

BlockEnvironment
Das BlockEnvironment ist für die Speicherung und Verwaltung des Geländes zuständig. Das Gelände besteht aus vielen einzelnen Blöcken (geometrischer Körper: Würfel), die 1x1x1 (X, Y, Z) in einem kartesischen Koordinatensystem einnehmen. Jeder Block hat eine eindeutige Position und kann sich nur auf einer Koordinate befinden. Die Koordinate wird durch einen Punkt mit ganzer Zahl für X, Y und Z repräsentiert ($P \in \mathbb{Z}^3$). Ein Block kann somit auch als Voxel [Voxel] betrachtet werden. Es gibt verschiedenen Blocktypen. Blöcke repräsentieren je nach Typ Böden, Gebäude, Wasser, Ressourcen oder andere vergleichbare Objekte. Sie können eingefügt, entfernt oder ersetzt werden.

UnitEnvironment
Das UnitEnvironment ist für die Verwaltung aller beweglichen Objekte zuständig. Jede im Spiel befindliche Einheit muss im UnitEnvironment registriert sein. Werden Einheiten eingefügt oder entfernt, informiert das UnitEnvironment über die GameWorld alle EventListener über das entsprechende Ereignis. Weiterhin wird der Status von allen Einheiten im UnitEnvironment gespeichert. In diesem Prototyp kann eine Einheit nur aktiv („active“) oder inaktiv („idle“) sein. Ist sie inaktiv, wird sie in einer Warteschlange

gespeichert und anderen Systemen auf Anfrage angeboten. Alternativ kann die Einheit auch aktiv sein, dann wird sie in einer Warteschlange gespeichert, die abhängig von dem Zeitpunkt, an dem sie ihren Auftrag erledigen wird, eingetragen. Einheiten werden nicht wie sonst üblich in jedem Spieltick geprüft [Minecraft] [Wilkes] [Hat07], sondern nur über vordefinierte Zeitpunkte, an denen sie ihren Auftrag erledigt haben und eine neue Aufgabe benötigen.

BuildingEnvironment
Im BuildingEnvironment werden alle Gebäude registriert. Da Gebäude das Gelände der Welt verändern (ein Weg wird beispielsweise durch die Platzierung eines Gebäudes unbegehbar) und sich auch über mehrere Koordinaten erstrecken können, wird ein Gebäude sowohl im BuildingEnvironment registriert, indem die Objektreferenz an jeder Koordinate gespeichert wird, die es belegt, als auch im BlockEnvironment. Für jedes Gebäude werden in diesem Fall Stellvertreter-Blocktypen für die jeweilig belegten Koordinaten platziert. Die Blöcke können das Gebäude selbst darstellen oder auch unsichtbare Blocktypen sein. Das Gebäude wird dann in einer visuellen Ansicht über ein gesondertes Model dargestellt. In diesem Prototyp werden Gebäude über sichtbare Blocke repräsentiert.

JobEnvironment
Ein Job bezeichnet einen Auftrag, der von einer Einheit abgearbeitet werden kann. Ein Auftrag kann beispielsweise der Abbau eines Blocks sein. Das JobEnvironment ist für die Verwaltung dieser Aufträge zuständig. Es speichert zudem grundlegende Informationen über Jobs, wie z.B. Prioritäten und an welcher Koordinate er sich befindet. Weiterhin gibt es ein ‚JobCenter' innerhalb des JobEnvironments, welches die Vergabe von Jobs übernimmt.

5.2.3 EnvironmentRenderer

Der EnvironmentRenderer ist für die visuelle Darstellung der Spielobjekte zuständig. Diese Komponente benötigt einen Grafikprozessor und Bibliotheken, um die Grafikeinheit anzusprechen. Der Einsatz des Renderers dient in dieser Arbeit als Debug- und Test-Hilfe. Ein Einsatz auf dem Serversystem ist nicht vorgesehen. Der EnvironmentRenderer ist somit eine optionale Komponente und wird auch als solche implementiert. Die Spielumgebung ist ohne die Komponente voll lauffähig.

5.2.4 Game-Mockup

Der Game-Mockup enthält eine einfache GameLoop [Gre90 S304 ff.], die den automatischen Ablauf der Simulation erlaubt. Zudem wird der Mockup für eine grafische Repräsentation der GameWorld benötigt und stellt rudimentäre Befehlseingaben zur Verfügung. Über Tasteneingaben kann hier beispielsweise die Kameraposition innerhalb der Spielwelt verändert oder der GameTick erhöht, und damit die Simulationszeit fortgeschritten werden.

5.3 Architektur des Verkehrsnetzes

Basierend auf den Anforderungen aus der Analyse, sowie den Eingrenzungen und Erläuterungen aus dem Anwendungsszenario wurden einige Konzept-Entscheidungen über das Verkehrsnetz erarbeitet. Diese werden im Folgenden aufgezählt und erläutert. Anschließend wird auf den Aufbau des Verkehrsnetzes eingegangen. Am Ende des Kapitels folgt eine Zusammenfassung des Konzepts.

5.3.1 Konzeptentscheidungen

Bei den Konzeptentscheidungen handelt es sich um Entscheidungen, die gemacht wurden, um verschiedene Eigenschaften des Prototyps festzulegen. Zu jeder Entscheidung existiert eine Erklärung, warum sie getroffen wurde.

Es ist nicht Aufgabe des Verkehrsnetzes einen optimalen Fluss der Transportmittel sicherzustellen.
Der Performanceaufwand für einen optimalen Fluss [WikiFlow] ist enorm. Abgesehen davon ist das Verkehrsnetz, die Koordination von Transporten sowie Transportrouten (z.B. für die Warenverarbeitung) ein essentieller Teil des Spielkonzepts. Das Spiel überlässt dem Spieler die Freiheit, sich sein eigenes Netz zu bauen und Erfahrungen zu sammeln.

Die Fairness im Verkehrsnetz ist nicht von Belang.
Das Verhungern eines Transportmittels (lange Inaktivität, obwohl es einen Job erledigen könnte) ist durchaus gewollt. Wie im vorhergehenden Punkt ist dies ein Aspekt der Verbesserung des Spielflusses und des Spielerlebnisses. Zudem kann es zum Teil auch Effektiver (aus Sicht des Spielers) und Effizienter (aus Sicht des Servers; Serverlast) sein, ein z.B. weit entferntes Transportmittel nicht zu verwenden, da sowohl mehr Zeit für die Wegsuche als auch für die Erfüllung des Auftrags benötigt wird. Das Vermeiden von verhungernden Transportmitteln erfordert, je nach Komplexität der aktuellen Spielwelt des Spielers, ein hohes Maß an Koordinationsdenken, Planung und Restrukturierung von Ressourcen.

Optimierung der Wegekalkulation durch Multithreading sollte nicht stattfinden.
Ein Prozessorkern [Multicore] des Hostsystems soll in der Lage sein, mehrere Spielerwelten zur selben Zeit zu verwalten. Ansonsten lässt sich ein Serversystem für die Verwaltung der Spielwelten nicht wirtschaftlich betreiben. Es macht daher im angestrebten Spielkonzept wenig Sinn, auf Multithreading oder -processing zu setzen. Anzumerken ist hier, dass bei einer Weiterentwicklung des Spielkonzepts, bei dem viele hundert Spieler in derselben Spielwelt agieren, der Einsatz von Multithreading bzw. -processing durchaus von Vorteil ist.

Im Verkehrsnetz kommt es zu keinen Staus mit Stillstand der Teilnehmer
Die Fahrzeuge im Verkehrsnetz können in jedem Moment ihre geplante Route zu einem Zielpunkt fortsetzen. Es ist ausgeschlossen, dass das Fahrzeug zum Stillstand kommt, bevor es sein Ziel erreicht hat. Das Verkehrsnetz benötigt dementsprechend auch keine Ampeln.

Dies ist die **wichtigste Konzeptentscheidung**. Sie schränkt die Umsetzungsvielfalt sehr stark ein und verhindert zudem die Möglichkeit, dass ein Stau entsteht, der das Verkehrsnetz lahmlegt.

Das Verkehrsnetz garantiert, dass zwei Fahrzeuge innerhalb des Netzes nicht mit einander kollidieren.
Um einen gewissen Grad an Realismus und Übersichtlichkeit zu erhalten, sollen Teilnehmer im Verkehrsnetz nicht durch einander hindurch fahren können. Denkbar wäre auch, Kollisionen nicht zu beachten. Beispielhaft dafür ist Anno1404 [WikiAnno], in dem das Verkehrsaufkommen aber sehr gering ist.

5.3.2 Integration des Verkehrsnetzes

Das in dieser Arbeit entwickelte Verkehrsnetz soll als Komponente analog zu den bisherigen Unterkomponenten (Environments) in das System integriert werden. Dabei soll die Kommunikation und Interaktion auf den gleichen Konzepten aufgesetzt werden. Der modulare Aufbau der Environments durch die Eventkommunikation erlaubt eine einfache Integration in die Spielumgebung. Wie in Abschnitt ***5.2.2*** beschriebenen, kann das Verkehrsnetz als Unterkomponente der GameWorld implementiert werden.

5.3.3 Graph mit Knoten und gerichteten Kanten

Das Verkehrsnetz wird als einfacher planarer gerichteter Graph modelliert [DGraph]. Ein einfacher planarer gerichteter Graph besteht aus Kanten und Knoten. Dabei können Knoten über Kanten verbunden werden. Gerichtet heißt, dass jede Kante im Graphen eine Richtung hat. Jede Kante hat somit einen Startknoten und einen Endknoten. Einfach heißt, dass keine Mehrfachkanten zwischen zwei Knoten bestehen und keine Schleifen existieren dürfen. Planar bedeutet, dass sich zwei Kanten niemals überkreuzen dürfen.

Im Verkehrsnetz hat jeder Knoten eine feste Position in der Spielwelt, und es darf nur ein Knoten pro Koordinate in der Welt existieren. Eine Kante entspricht der Strecke zwischen zwei Knoten. Dabei sind in diesem Prototyp nur Wege erlaubt, die parallel zu den Achsen verlaufen und keinen Höhenunterschied aufweisen. Für einen Knoten an Position P gilt, dass $P \in \mathbb{Z}^3$ (Ein Knoten kann nur an ganzzahligen Koordinaten plaziert werden). Knoten und Kanten zusammen ergeben eine Straße. In ***Abbildung 2*** ist (von oben nach unten) ein einzelner Knoten(A), eine einzelne Kante, eine Kante zwischen zwei direkt aneinandergrenzenden Knoten (B,C) und eine Kante zwischen zwei Knoten zu erkennen, die über fünf Blöcke gespannt ist (D,E). Die Pfeile geben dabei die Richtung der Kanten an.

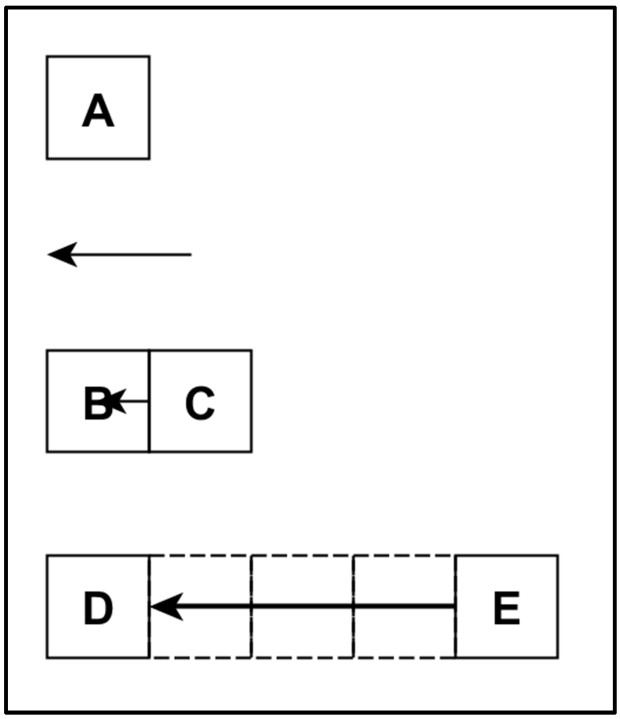

Abbildung 2: Straßenbeschreibung

Wird ein Knoten (C), wie in ***Abbildung 3*** zu sehen ist, zwischen zwei existierende Knoten (A,B) eingefügt, so muss die Kante zwischen A und B entfernt, und zwei neue gespannt werden (B → C, C → A).

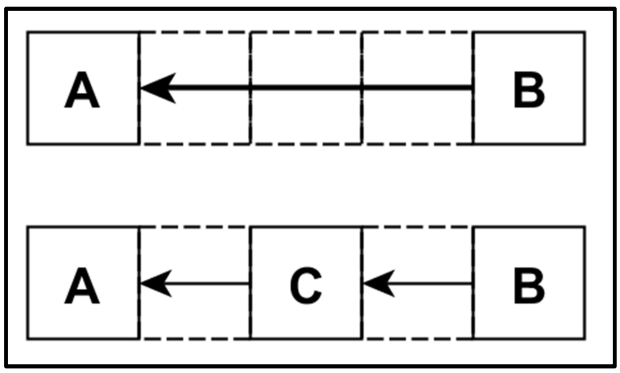

Abbildung 3: Einfügen eines Knoten zwischen zwei Knoten

Eintrittspunkt und Austrittspunkt

Eintrittspunkte dienen Fahrzeugen als Zugang zu Kanten (Dem Straßenabschnitt zwischen zwei Knoten). Fahrzeugen ist es nur erlaubt, über Knoten auf eine Kante zu gelangen. Je nach Straßentyp kann ein Knoten (Kreuzungspunkt) mehrere Eintritts- und Austrittspunkte haben. Ein durchgestrichener Pfeilansatz bezeichnet dabei einen unpassierbaren Weg. Jeder Knoten hat maximal vier Punkte (einen je Seite), an denen er mit Nachbarknoten verbunden werden kann. Die ***Abbildung 4*** zeigt (v.l.n.r.):

- Einen einzelnen Knoten ohne Eintritts oder Austrittspunkte
- Einen Knoten, der genau einen Ausgangspunkt zu einem Nachbarknoten besitzt
- Einen Knoten, der genau einen Eingangspunkt von einem Nachbarknoten besitzt

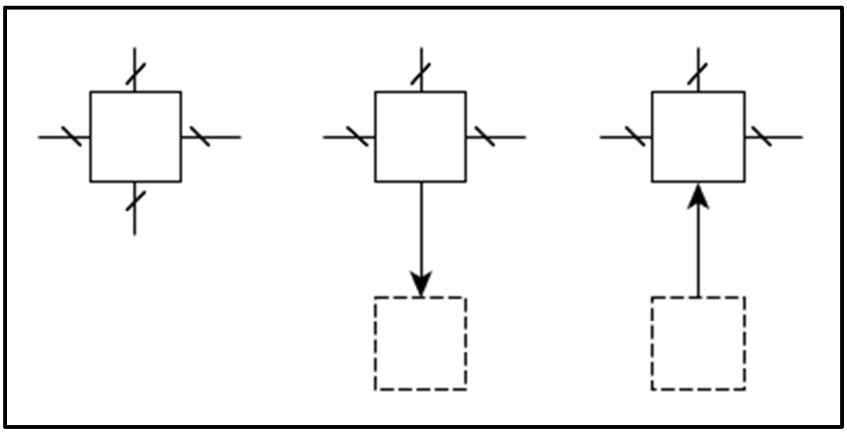

Abbildung 4: Verschiedene Verbindungspunkte eines Knoten

Straße

Eine Straße besteht, sobald ein Knoten im Graphen existiert. In dieser Arbeit wird ein Knoten als Kreuzungspunkt bezeichnet. In ***Abbildung 5*** ist eine Einbahnstraße zu sehen, die aus drei Knoten bzw. Kreuzungspunkten besteht. Es ist nur möglich von C über B nach A zu fahren.

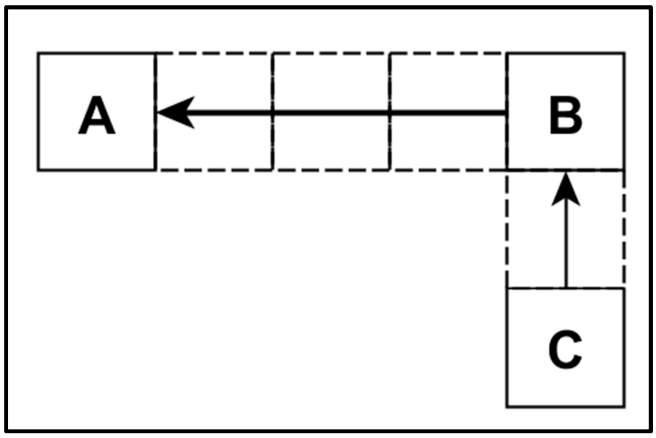

Abbildung 5: Einbahnstraße mit drei Kreuzungspunkten

Zur Darstellung von komplexeren Straßenstrukturen, wie beispielsweise einer Kreuzung, sind mehrere Knoten und Kanten nötig. Die ***Abbildung 6*** zeigt eine Kreuzung, die über vier Knoten (A,B,C,D) und acht weiteren Knoten in der Umgebung, sowie vier Kanten zwischen den Knoten der Kreuzung und acht weiteren Kanten zu den Umgebungsknoten, realisiert werden kann. Jeder der vier Kreuzungspunkte der Kreuzung hat zwei Eintrittspunkte und zwei Austrittspunkte.

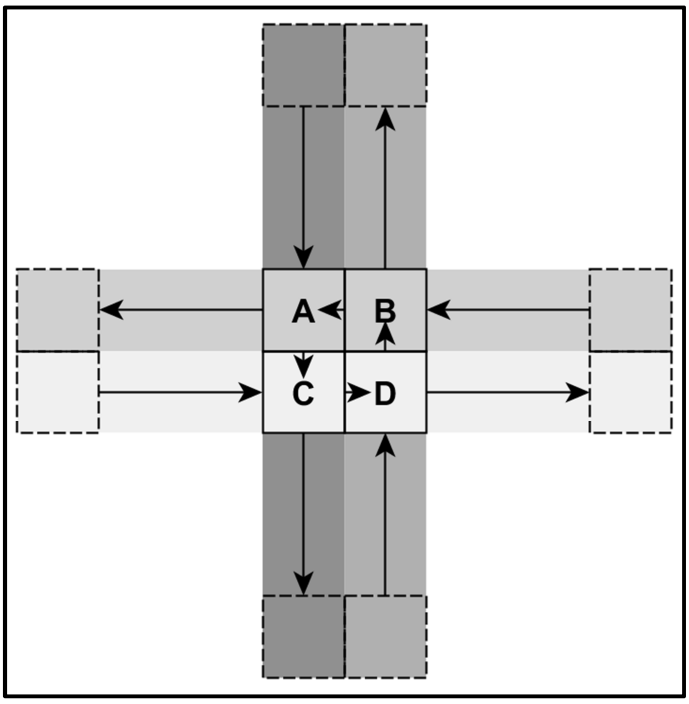

Abbildung 6: Definition einer Kreuzung im Straßensystem

Beschreibung der Straßennutzung

Eine Straße erlaubt es den Fahrzeugen, über diese zu fahren. Dabei soll garantiert werden, dass das Fahrzeug während der Fahrt im Verkehrsnetz nicht in Kontakt (Kollision) mit anderen Fahrzeugen gerät. Zudem soll ein Stau, der das Anhalten des Fahrzeuges erzwingt, ausgeschlossen werden. Jedem Fahrzeug soll demnach auch garantiert werden, dass es innerhalb des Verkehrsnetzes niemals anhalten muss. Die Einhaltung der Vorfahrtsregel „Rechts vor Links“ [Vorfahrtsregel] ist demnach nicht mehr nötig, da es niemals dazu kommen wird, dass ein Fahrzug für ein anderes stoppen muss.

5.3.4 Garantierte Kollisionsfreiheit

Die bisher vorgestellten Verwaltungskonzepte des Verkehrsnetzes über einen Graphen mit gerichteten Kanten erlauben den Fahrzeugen, sich in diesem System zu bewegen und sich dabei an simple Regeln zu halten. Es garantiert aber noch keinen kollisionsfreien Verkehr. Um eine Garantie zu erlangen und dabei möglichst performant zu sein, wird stark auf

Vorberechnung gesetzt. Das Verkehrsnetz kann hier als isoliertes System angesehen werden. Ein Fahrzeug darf dieses System nur benutzen, wenn einige Vorbedingungen erfüllt sind.

- Das Fahrzeug muss einen festen Start und Zielpunkt aufweisen.
- Das Fahrzeug darf das Verkehrsnetz nur über den Start und Zielpunkt betreten und verlassen.
- Das Fahrzeug muss in der Lage sein, über das Verkehrsnetz vom Start zum Ziel zu gelangen.
- Das Fahrzeug muss in der Lage sein, ohne Pause vom Start zum Ziel zu gelangen.
- Das Fahrzeug darf, während es im Verkehrsnetz ist, nicht mit anderen Fahrzeugen des Verkehrsnetzes kollidieren.

Zur Einhaltung dieser Vorbedingungen ist eine besondere Informationsstruktur nötig. Da ein Fahrzeug auf dem Weg von einem Start- zu einem Zielpunkt prinzipiell jederzeit mit einem anderen Fahrzeug kollidieren könnte, muss das Verkehrsnetz Informationen anbieten, um Zusammenstöße unter Teilnehmern zu verhindern.

- Das Verkehrsnetz muss eine Aussage darüber treffen können, ob ein Fahrzeug von einem Startpunkt zum Zeitpunkt **t** zu einem Zielpunkt zum Zeitpunkt **t‘** gelangen kann.
- Das Verkehrsnetz muss sicherstellen, dass ein Fahrzeug an jedem Knoten während des Zeitsegments seiner Überquerung nicht mit anderen Fahrzeugen kollidiert.
- Das Verkehrsnetz muss sicherstellen können, dass die Durchquerung eines Wegs für ein Fahrzeug ohne Kollisionen mit anderen Fahrzeugen möglich ist.

In dieser Arbeit wird zunächst nur von Fahrzeugen ausgegangen, die sich alle mit der gleichen konstanten Geschwindigkeit im Verkehrsnetz bewegen. Um die Kollisionsfreiheit im Verkehrsnetz zu garantieren wird jeder Knoten mit einer Datenstruktur ausgestattet, die Zeitsegmente belegen kann. Der Knoten speichert dabei den Eintrittszeitpunkt, den Austrittszeitpunkt und die Referenz eines Verkehrsteilnehmers, der diesen Knoten benutzen möchte. Einem Fahrzeug ist es nur erlaubt, einen Knoten zu passieren, wenn es sich für exakt den Zeitraum, den es benötigt, um den Knoten zu überqueren, auch registriert hat.

Ein Fahrzeug kann nicht direkt mit Knoten kommunizieren. Es kann lediglich bei dem Verkehrsnetz eine Anfrage auf eine Route stellen. Diese Anfrage muss einen Start- und einen Zielpunkt enthalten, sowie einen Startzeitpunkt. Sofern eine Route gefunden werden konnte, bietet das Verkehrsnetz dem Fahrzeug diese Route mit exakt vorberechneten Zeitpunkten an. Die Route enthält jeden Zwischenpunkt (Knoten) vom Start zum Zielpunkt, sowie die Eintritts und Austrittszeiten, die dem Fahrzeug zugesprochen werden konnten. Das Fahrzeug kann diese Route nun beim Verkehrsnetz registrieren. Erfolgt dieser Aufruf, werden bei allen Zwischenknoten die Zeitsegmente registriert und stehen für keinen weiteren Teilnehmer im Verkehrsnetz mehr zu Verfügung. Der Graph des Verkehrsnetzes ist demnach hoch variabel, da bei jeder Registrierung sich der Graph in Abhängigkeit von

registrierten Zeitsegmenten für weitere Anfragen anders darstellt. ***Abbildung 7*** zeigt eine mögliche Belegung von zwei Knoten im Verkehrsnetz. **S[0,100, unit1]** steht dabei für **Zeitsegment[Starttick, Endtick, Referenz]**.

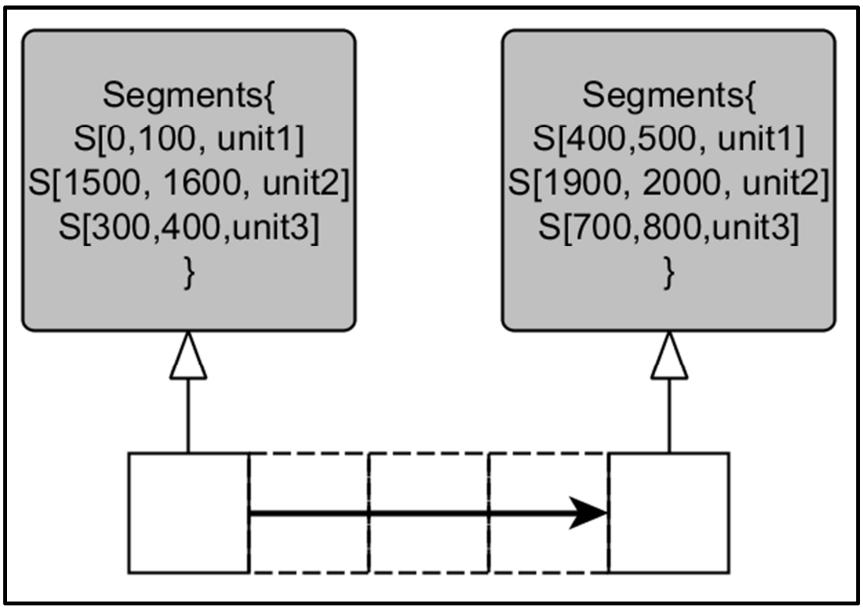

Abbildung 7: Mögliche Einheitenregristrierung in zwei Knoten

Zusammenfassung

Das vorgestellte Konzept des Spielwelt-Prototyps erlaubt, die Integration des erarbeiteten Verkehrsnetzkonzepts. Dabei wurde besonderen Wert auf eine spätere Erweiterbarkeit der Komponenten gelegt. Speziell sei hier auf die Erweiterung des Verkehrsnetzes auf Flug-, Schienen- und Wasserwege hingewiesen.

Das vorgestellte Verkehrsnetz ist eine sehr spezielle Form eines Verkehrsnetzes. Die Fahrzeuge haben keine Entscheidungsfreiheit über den Weg, den sie fahren. Es können keine Staus entstehen, da die Wege vom Start bis zum Ziel durchgeplant und garantiert sind. Weiterhin werden Unfälle in diesem Verkehrsnetz ausgeschlossen.

Den außerordentlichen Vorteil, den dieses System bietet, ist die kollisionsfreie Fahrt von Fahrzeugen innerhalb des Verkehrsnetzes ohne Kollisionsprüfungen zur Laufzeit. Ein übliches Verkehrsnetz muss in jedem Zeitschritt Kollisionsprüfungen durchführen [Nag99] [Hat07]. Die dabei entstehende Grundlast (also die Kollisionsprüfung zwischen Fahrzeugen) steigt mit der Zahl der Fahrzeuge an. Diese Grundlast ist für ein Serversystem eine starke Einschränkung. Es geht ein Teil der Prozessorzeit in jedem Tick verloren. Das Ziel des hier vorgestellten Konzepts ist es, diese Grundlast auf das System zu entfernen. Es findet lediglich eine Belastung bei der Wegfindung und Knoten-Registrierung statt. Ist dieser Prozess abgeschlossen und das Fahrzeug auf dem Weg von seinem Start- zum Zielpunkt, fallen keine Berechnungen mehr an. Für das Serversystem muss lediglich eine Warteschlange geführt werden, in der das Fahrzeug und seine Ankunftszeit festgehalten wird, damit dem Fahrzeug weitere neue Aufträge erteilt werden können.

6 Realisierung

In diesem Kapitel wird zunächst das Konzept der Spielumgebung umgesetzt. Anschließend wird das Verkehrsnetz darin integriert. Für die Realisierung wurde in dieser Arbeit die Programmiersprache C# [CSharp] gewählt. Für die grafische Oberfläche ist das XNA Framework [XNA] [Gre09 S26 f.] verwendet worden. XNA vereinfacht die Entwicklung von Spielen. Es enthält Werkzeuge zur Spieleentwicklung und -management [WikiXNA]. Als Entwicklungsumgebung ist das Visual Studio 2010 [VisualStudio] zum Einsatz gekommen. Bei der Implementierung wurde auf die Nutzung von Delegates [Delegates] und Events [Events] gesetzt.

6.1 Zielsetzung der Realisierung

Das Ziel der Realisierung ist zunächst die Umsetzung des Konzepts für die Spielumgebung und das Verkehrsnetz. Dabei sollen möglichst performante Datenstrukturen und Implementierungen verwendet werden. Weiterhin soll die Implementierung eine grobe Abschätzung der Leistungsfähigkeit des Systems (siehe Kapitel *7*) ermöglichen. Weitere Ziele sind die Testbarkeit des Prototys und der damit einhergehende Determinismus des Systems. Hier ist vor allem die Trennung der grafischen Repräsentation von den logischen Typen und Klassen wichtig, um das einfache Testen über einen nullView-Client [MMDG2 S178 ff.] zu ermöglichen. Die Trennung erlaubt zudem den späteren Betrieb auf Serversystemen ohne grafische Recheneinheit.

Die grafische Repräsentation der Spielwelt wird Implementiert um visuelle Tests zu ermöglichen. Bei diesem Teil der Arbeit wird kein Wert auf die Performanz gelegt. Die Implementierung erhebt keinen Anspruch auf gute Benutzbarkeit oder Zuverlässigkeit.

6.2 Realisierung der Spielumgebung

Im ersten Teil der Arbeit wird erst auf die GameWorld Fassade eingegangen und die verwendeten Code-Konzepte erläutert, anschließend folgt eine Erläuterung der Komponenten des Spiels. Speziell wird auf die verwendeten Techniken zur Performanceoptimierung eingegangen, sowie auf potentiell bessere Methoden verwiesen, die in der Arbeit nicht mehr implementiert werden konnten.

6.2.1 Das Schematic

Der Begriff ‚Schematic' wurde den Begriffen Pattern oder Model vorgezogen, da diese in der Informatik bereits für bekannte Konzepte verwendet werden. Er ist als Schema, Bauplan oder Muster zu verstehen.

Es dient als Änderungsanfrage an die Spielumgebung, ohne eine genauere Definition über die Art der Änderung vorzunehmen. Ein Schematic könnte beispielsweise einen Koordinatenpunkt, einen Block und die Anweisung diesen Block an der Koordinate zu

platzieren enthalten. Ein komplexeres Schematic wäre das Einfügen eines Gebäudes, welches aus Blöcken, Gebäudepositionen und einem Gebäudeobjekt besteht.

6.2.2 GameWorld Fassade

Die GameWorld stellt im aktuellen Prototyp die Fassade für alle grundlegenden Anfragen dar. Sie vereint die Funktionalität der Unterkomponenten in einem aufrufbaren Interface. Die Anfragen und Aufrufe werden dabei über Events an die Unterkomponenten weitergeleitet. Soll ein Objekt in die Spielwelt eingefügt werden, wird beispielsweise die Methode AddSchematic (siehe ***Abbildung 8***) aufgerufen.
Die Methode teilt sich in zwei Teile. Im ersten Teil (CanOrderRequest) wird das Schematic an alle Unterkomponenten weitergeleitet und von diesen geprüft (z.B. ob ein Block an der Koordinate eingefügt werden darf). Dabei darf das Schematic nur geprüft und nicht verändert werden. Als Ergebnis des Aufrufs kommt eine Menge an Prüfergebnissen der Unterkomponente zurück. Jeder Empfänger des Events muss seine Prüfergebnisse (ResultState) für das Event angeben. Sind alle Ergebnisse der Prüfung in Ordnung (ResultState.LogicPassed), so wird der zweite Teil der Methode ausgeführt. Dabei wird dasselbe Schematic erneut an alle EventListener versendet und die enthaltenen Änderungen ausgeführt.

Die Teilung in zwei separate Teile erfolgt auf Grund der vielen Sub-Komponenten, die möglicherweise zu einem späteren Entwicklungszeitpunkt noch hinzugefügt oder erweitert werden. Es erfolgt daher eine Validierung der Eingaben, bevor eine Komponente Änderungen vornimmt. Beispielhaft seien hier eine Rechte-Komponente oder eine Währungskomponente genannt, die die Platzierung eines Gebäudes verhindern könnten, obwohl das Gebäude an dem gewählten Ort prinzipiell platziert werden könnte. Das Event wird bei der logischen Prüfung immer durch alle EventListener gereicht, auch wenn eine Unterkomponente bereits einen Fehler zurückgibt. Dies erlaubt in einem späteren Entwicklungsstadium bessere Fehlermeldungen.

```
public void AddSchematic(Point3 p, ISchematic schematic)
{
    // Stage 1, get allowance from all EventListeners/SubComponents
    var param = RequestFactory.AddSchematicRequest(p, schematic);
    var result = CanOrderRequest(param);
    // Stage 2, actually add the schematic information
    if (result.WorstResult == ResultState.LogicPassed)
        OrderRequest(param);
}
```

Abbildung 8: Die AddSchematic Methode in der GameWorld

Die einzelnen Komponenten melden sich auf die Events der GameWorld an. Wird ein CanOrderRequest-Event von der GameWorld geworfen, muss die jeweilige Sub-Komponente prüfen, ob es erlaubt ist, diesen Auftrag auszuführen. Dazu werden die Request-Argumente, die für die jeweilige Komponente von Bedeutung sind, validiert. ***Abbildung 9*** zeigt die CanOrderHandler Methode des BlockEnvironemts. Die Aufgabe der Methode ist es zu prüfen, ob Blöcke eingefügt, gelöscht oder ersetzt werden dürfen. Der ResultState könnte hier z.B. durch den Versuch einen nicht existierenden Block zu entfernen auf den Wert „LogicFailed“ gesetzt werden und zu einem Abbruch führen.

Sind alle Validierungen erfolgreich, so schreibt die Komponente den ResultState „LogicPassed“ in die CanRequestArgs, und der Aufruf der OrderRequest-Methode ist möglich. Dieser Ablauf ist im Sequenzdiagram ***Abbildung 10*** sichtbar.

```
private void CanOrderHandler(object o, CanRequestArgs requestArgs)
{
    var resultState = ResultState.LogicPassed;
    if (requestArgs.HasAddBlocks)
    {
        var state = CanAddBlocks(
            (IEnumerable<Tuple<Point3, Block>>) requestArgs.GetModifications(ModificationId.AddBlocks));
        resultState = ResultMethods.WorstState(resultState, state);
    }
    if (requestArgs.HasRemoveBlocks)
    {
        var state = CanRemoveBlocks(
            (IEnumerable<Tuple<Point3, Block>>)requestArgs.GetModifications(ModificationId.RemoveBlocks));
        resultState = ResultMethods.WorstState(resultState, state);
    }
    if (requestArgs.HasChangeBlocks)
    {
        var state = CanChangeBlocks(
            (IEnumerable<Tuple<Point3, Block>>)requestArgs.GetModifications(ModificationId.ChangeBlocks));
        resultState = ResultMethods.WorstState(resultState, state);
    }
    // Add a logical Resultstate
    requestArgs[ResultIdentifier.BlockEnv] = resultState;
}
```

Abbildung 9: Validierung eines Auftrags im BlockEnvironment

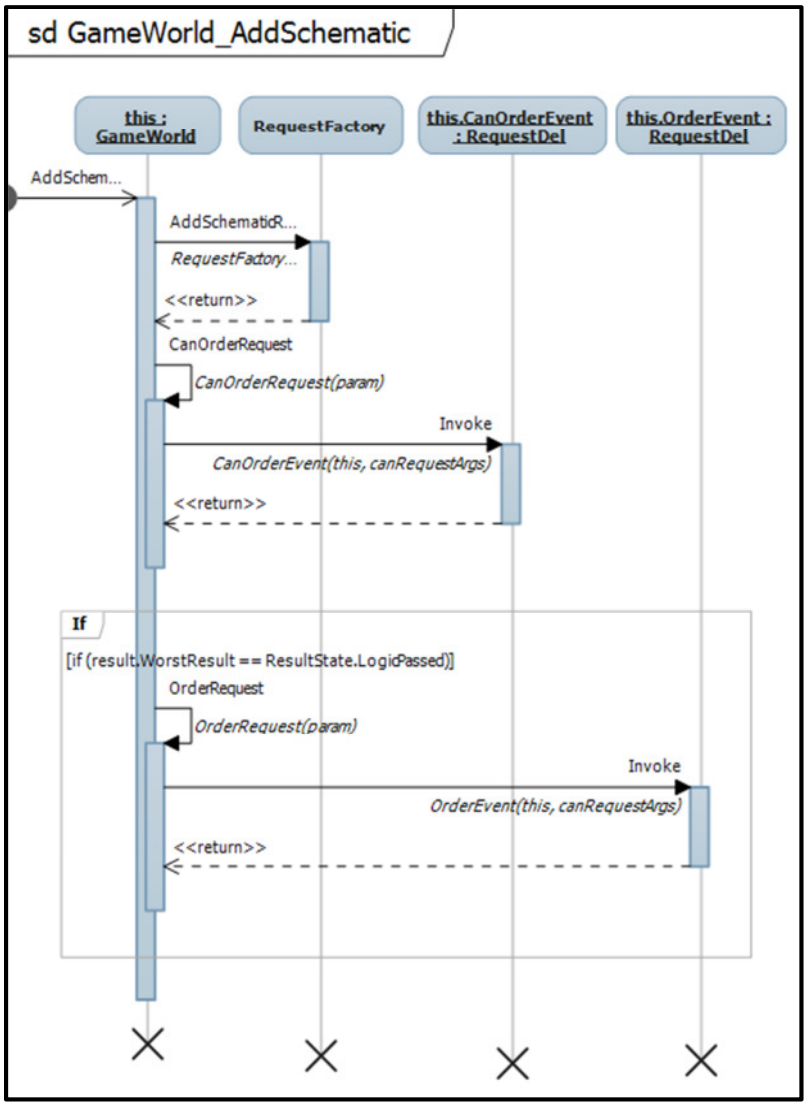

Abbildung 10: Sequenzdiagram der AddSchematic Methode

6.2.3 Komponentenimplementierungen

Im Folgenden werden kurz die einzelnen Unterkomponenten der Spielumgebung erläutert. Diese werden als Environments bezeichnet und kapseln jeweils bestimmte Aufgabenfelder.

BlockEnvironment

Das BlockEnvironment ist für die Verwaltung und Anpassung aller Blöcke in der Spielwelt zuständig. Es erhält Anfragen über Events und nimmt entsprechende Änderungen an der BlockMap vor. Die BlockMap speichert alle Blöcke der Spielwelt in einer in kleinere Regionen unterteilte Datenstruktur. Die BlockMap besteht aus BlockRegionen, welche wiederum aus BlockContainern bestehen. Ein BlockContainer ist ein dreidimensionales Array aus Blöcken. Die BlockRegion fasst eine über die Spielkonfiguration einstellbare Anzahl an BlockContainern in sich zusammen. Die Container werden in einem Dictionary gespeichert, in der der Key die Position des BlockContainers ist. Die BlockMap fasst alle BlockRegionen

der Spielwelt zusammen. Auch hier wird ein Dictionary mit der Position der BlockRegion als Key verwendet. In ***Abbildung 11*** ist der Aufbau der BlockMap grafisch dargestellt.

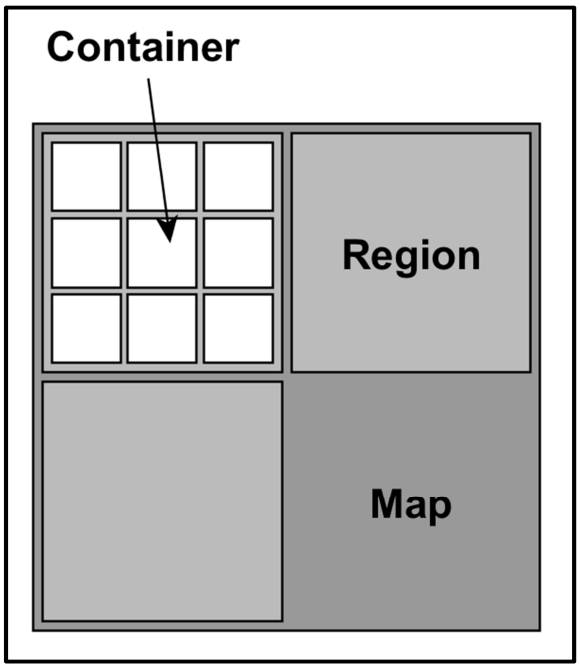

Abbildung 11: Zweidimensionale Ansicht der BlockMap-Struktur

Die Implementierung wurde primär gewählt, um in einem späteren Entwicklungsstadium der Software daraus Vorteile zu ziehen. Die Struktur erlaubt sehr große Welten (Integer.MinValue bis Integer.MaxValue). BlockRegionen und BlockContainer können einer Spielwelt einfach hinzugefügt werden. Es besteht die Möglichkeit sie mit TimeStamps der letzten Änderung auszustatten. Dies erlaubt es den Clients, Regionen und Container zu speichern und aus ihrem eigenen Cache zu laden, sollte der TimeStamp ihrem eigenen entsprechen. Zudem ist es dem Serversystem möglich, viele Informationen vor dem Client zu verstecken. Ein böswilliger Spieler (Cheater) könnte probieren, sich einen Vorteil im Spiel zu verschaffen, indem er normalerweise unsichtbare Blöcke im Untergrund (Ressourcen) sichtbar macht. Mit dieser Information könnte er viel effektiver spielen als andere Teilnehmer. Sendet der Server allerdings nur BlockRegionen und BlockContainer, die vom Spieler unmittelbar benötigt werden, wird diese Möglichkeit stark eingeschränkt.

BuildingEnvironment

Die Implementierung des BuildingEnvironment orientiert sich am Aufbau des BlockEnvironments. Es besteht aus einer BuildingMap, welche BuildingRegionen enthält, die jeweils für einen Teil der Spielkarte die Verwaltung der Gebäude übernehmen. An Stelle der BlockContainer kommt allerdings ein Dictionary zum Einsatz, um Speicherplatz zu sparen, da jede BuildingRegion wesentlich weniger Gebäude-Einträge enthalten wird, als sie Platz bietet. Eine BlockRegion hat hingegen an prinzipiell jeder Koordinate einen Blocktypen.

UnitEnvironment
Das UnitEnvironment soll sich an die gleiche Struktur halten, die auch im BlockEnvironment verwendet wird (Map mit Regionen). Da es sich bei Units um bewegliche Objekte handelt, müssen diese bei der Überschreitung einer Regionsgrenze von einer Region in die andere übertragen werden. In dieser Arbeit wird allerdings eine vereinfachte Struktur verwendet, die dem Prototyp genügt. Dabei werden alle Einheiten immer in der Region verwaltet, der sie erstmals zugewiesen wurden. Eine Auswirkung hat dies im diesem Stadium der Arbeit nur auf den EnvironmentRenderer, da dieser nun auch Einheiten zeichnet, die außerhalb des Sichtfeldes des Betrachters liegen.

JobEnvironment
Auch das JobEnvironment orientiert sich an der Implementierung des BuildingEnvironments und nutzt auf der Ebene der Regionen aktuell HashSets zur Speicherung von Auftragskoordinaten. Es wird in der Implementierung derzeit nur die Koordinate gespeichert, da es nur eine Art von Job gibt. In einer erweiterten Version würde an Stelle des HashSets ein Dictionary eingesetzt werden, um den jeweiligen Auftrag speichern zu können.

Jobs können zudem unterschiedliche Prioritäten haben, Aufträge mit einer höheren Priorität werden immer als erstes abgearbeitet. Weiterhin werden Aufträge, die von einer Einheit angenommen wurden, in einer gesonderten Datenstruktur ‚occupiedJobs' gespeichert, bis sie abgearbeitet wurden (***Abbildung 12*** zeigt diese Datenstruktur). Dies ist nötig, um die aktuell koordinatenabhängigen Jobs nicht erneut zu erstellen und zu vergeben, falls eine Einheit noch dabei ist, den Auftrag an der Koordinate zu erledigen.

```
private readonly SortedList<JobPriority, HashSet<Point3>> jobs;
private readonly SortedList<JobPriority, HashSet<Point3>> occupiedJobs;
```

Abbildung 12: Nach Priorität sortierte Jobs in einer JobRegion

CubedElements
Die CubedElements Komponente kapselt benötigte Klassen für die grafische Repräsentation der Spielwelt. Das Projekt baut auf das XNA Framework auf und stellt dem EnvironmentRenderer die Texturen und Effekte zur Verfügung. Dabei gibt es eine klare Trennung der Komponenten, um die Gameworld auch ohne das XNA Framework betreiben zu können. Alle grafischen Komponenten, somit auch die CubedElements Komponente, sind optional.

Effekte und Texturen sind in einem Content-Projekt gekapselt, welches von XNA zur Verfügung gestellt wird. Modell-Klassen, die für das Rendern von Objekten nötig sind, werden in einer eigenen Komponente „PrimitiveRenderTypes" gekapselt.

EnvironmentRenderer
Der EnvironmentRenderer verwaltet Informationen und Objekte, die zur grafischen Darstellung von Blöcken, Einheiten, Gebäuden und Jobs nötig sind. Die Komponente wird durch Events über Spieländerungen informiert. Bei der grafischen Darstellung wird aufgrund

der hohen Anzahl an gleichen Objekten auf Geometry Instancing bzw. Hardware Instancing [Gre09 406 f.] gesetzt. Alle Objekte werden mit einer eindeutigen einzelnen Position modelliert (Voxel [Voxel]) und als Würfel dargestellt. In ***Abbildung 13*** ist eine beispielhafte grafische Szene zu sehen. Der gelbe Punkt repräsentiert ein Fahrzeug und kann sich bewegen. Der bläuliche Block mit dem grünen Auto ist ein Block, der als Auftrag ausgeschrieben wurde und von einer Einheit bereits zur Abarbeitung angenommen wurde. Der rötliche Block ist ein noch ausstehender Auftrag. Die weißen Blöcke mit den Pfeilen sind die grafische Repräsentation von Knotenpunkten im Verkehrsnetz. Ihre Ausgangsfahrtrichtung wird mit den Pfeilen angegeben. In diesem Beispiel handelt es sich demnach um eine Einbahnstraße. Die grünen Blöcke sind einfache Grass-Bodenblöcke ohne direkten Nutzen. Der rote Strich ist eine Debug-Information und markiert den Punkt(0, 0, 0) im Koordinatensystem. Er verläuft auf der Y-Achse in positiver Richtung.

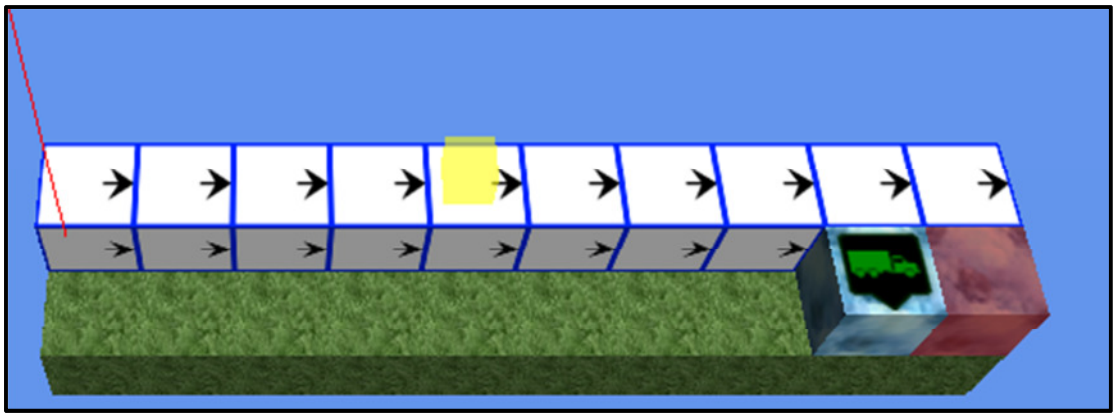

Abbildung 13: Beispiel der grafischen Darstellung

Zusammenfassung

Die vorgestellte Spielumgebung erlaubt es, eine Spielwelt zu generieren. Innerhalb der Spielwelt sind prinzipiell alle Objekte manipulierbar. Es ist möglich, Einheiten und Gebäude einzufügen und Aufträge zu erstellen. Diese Aufträge können von Einheiten angenommen und abgearbeitet werden. Im Folgenden soll nun erläutert werden, wie das Verkehrsnetz in das System integriert wird und wie Einheiten über das Verkehrsnetz zu ihren Aufträgen gelangen. Es wird zudem ausgeführt, welche Klassen und Datenstrukturen in das Grundsystem integriert werden, um eine effiziente und performante Verwaltung der Einheiten, Aufträge und Straßen zu unterstützen.

6.3 Integration des Verkehrsnetzes

Die im Konzept beschriebene Komponentenstruktur in Verbindung mit EventListenern erlaubt eine leichte Erweiterung der Spielumgebung um eine Verkehrskomponente. Diese soll im Folgenden beschrieben werden. Zunächst wird ein Szenario beschrieben, in dem ein möglicher Einsatzzweck des Verkehrsnetzes aufgezeigt wird. Anschließend erfolgt eine detaillierte Erläuterung der Umsetzung, in der speziell die Umsetzung von performancerelevanten Teilen angesprochen wird.

6.3.1 Szenariobeschreibung

Das Szenario baut auf dem in Kapitel 3 genannten abstrahierten Szenario auf. Das in diesem Prototyp umgesetzte Szenario ist eine einfache Spielwelt, in der einige Blöcke existieren, die als Jobs markiert sind. Diese Blöcke können von den Einheiten in der Spielwelt abgebaut werden. Die Einheiten müssen dabei über das Verkehrsnetz in die direkte Umgebung der spezifizierten Blöcke gelangen, um diese abbauen zu können. Für den Abbauvorgang selbst muss das Fahrzeug das Verkehrsnetz verlassen und auf die Koordinate des Auftrags fahren.

Dieses relativ simple Szenario kann genutzt werden, um Erkenntnisse über das System zu gewinnen. Die variable Spielweltgröße bietet eine Möglichkeit, die Zahl an Einheiten, Aufträgen und Straßen zu erhöhen. Die GameWorld kann zudem mehrfach Instanziiert werden, um eine Vielzahl an gleichzeitigen Nutzern zu simulieren.

6.3.2 Knoten

Jeder Knoten im Graph des Verkehrsnetzes (TrafficSystem) besitzt Informationen über die Fahrtrichtungen zu seinen Nachbarn. Diese können unpassierbar, eingehend oder ausgehend sein. Besitzt der Knoten keinen Nachbarn in einer Richtung, kann die Fahrtrichtung auch gesetzt sein, der Weg bleibt aber unpassierbar. Weiterhin besitzt jeder Knoten eine Baumstruktur, in der die Zeitsegmente gespeichert werden, die Einheiten zum Überqueren der Kreuzung registriert haben (siehe Abschnitt ***5.3.4***). Bei der Baumstruktur ist eine hohe Performanz wichtig, weswegen eine Implementierung [LLRB2] des LLRB-Tree (Eine Variante des Red-Black-Tree) verwendet wurde. Der LLRB-Tree besitzt eine Komplexität von O(log n) für Suchanfragen, das Einfügen und das Entfernen von Einträgen [LLRB1]. Diese Eigenschaft ist für die hochdynamischen Knoten sehr wichtig.

LLRB-Tree Implementierung

Das LLRB-Tree speichert Werte an Hand eines Keys. Es wurde entwickelt, um einzelne Keys-Value-Paare zu finden. Die GetValueForKey Methode liefert bei einem Aufruf den gespeicherten Wert zu einem Key zurück oder wirft eine `KeyNotFoundException`, wenn der Key nicht gefunden werden konnte. ***Abbildung 14*** zeigt den Code der Methode.

```
public Tuple<TKey, IMovableGameObject> GetValueForKey(TKey key)
{
    Node node = GetNodeForKey(key);
    if (null != node)
    {
        return node.Value;
    }
    throw new KeyNotFoundException();
}
```

Abbildung 14: Implementierung der GetValueForKey Methode

Diese Methode ist für die Knoten nicht nutzbar, da sie nur Zeitsegmente findet, die exakt an dem übergebenen Zeitpunkt beginnen. Das bedeutet, dass ein Zeitsegment, dessen Zeitintervall den übergebenen Zeitpunkt einschließt, aber nicht genau an diesem beginnt, nicht zurück geliefert wird. Anstelle der GetValueForKey-Methode, die zu einem gegebenen Schlüssel den Wert zurückgibt, wurde eine Methode (GetNearestNeighbours) implementiert und verwendet, die die dichtesten Nachbarwerte zurück liefert (siehe ***Abbildung 15***). Die Methode liefert in jedem Fall ein Ergebnis, welches drei Werte enthält. Bei den Werten handelt es sich um Knoten des Baums. Die drei zurückgelieferten Werte sind der Knoten mit dem dichtesten kleineren Zeitintervall, der Knoten mit dem dichtesten größeren Zeitintervall und falls vorhanden, den Knoten mit dem gesuchten Startzeitpunkt selbst.

Ein Zeitsegment kann in einem Knoten registriert werden, wenn folgende Bedingungen zutreffen:

- Der gesuchte Key selber ist nicht im Baum vorhanden
- Der Startzeitpunkt ist größer (später) oder gleich dem Endzeitpunkt des vorhergehenden Eintrags
- Der Endzeitpunkt ist kleiner (früher) oder gleich dem Startzeitpunkt des folgenden Eintrags

In ***Abbildung 15*** ist die Implementierung der GetNearestNeighbours-Methode zu erkennen. Sie ist im Gegensatz zu der GetValueForKey komplexer und auch aufwendiger in der Durchführung, da sie mehrere Werte zurückliefern muss. **_minNode** und **_maxNode** sind feste Ausgangsknoten, die Integer.MinValue bzw. Integer.MaxValue als Wert haben. Die Erweiterung des LLRB-Trees erlaubt es, in den Knoten Zeitintervalle abzufragen und zu registrieren. Implementiert ist dies über zwei Methoden in den Knoten. **IsFree** und **Register**, wobei die IsFree Methode keine Änderungen am Graphen vornimmt. Sie prüft, ob ein Knoten zu einem übergebenen Zeitfenster belegt ist. Die Register Methode erlaubt die Anmeldung an dem jeweiligen Knoten auf ein gegebenes Zeitfenster. Es findet keine erneute Prüfung der Eingabewerte statt, diese Prüfung muss das Verkehrsnetz zuvor über den Aufruf der IsFree Methode sichergestellt haben.

```
public Tuple<Node, Node, Node> GetNearestNeighbours(TKey key)
{
    Node current = _rootNode;
    Node closestSmallerValue = _minNode;
    Node closestBiggerValue = _maxNode;
    // Get as close as we can to our wanted spot
    while (null != current)
    {
        int comparisonResult = _keyComparison(key, current.Key);
        if (comparisonResult < 0)
        {
            if (_keyComparison(current.Key, closestBiggerValue.Key) < 0) closestBiggerValue = current;
            current = current.Left;
        }
        else if (0 < comparisonResult)
        {
            if (_keyComparison(current.Key, closestSmallerValue.Key) > 0) closestSmallerValue = current;
            current = current.Right;
        }
        else
        {
            // We found our actual key, return the result as we wont be able to register it anyways
            return new Tuple<Node, Node, Node>(closestSmallerValue, current, closestBiggerValue);
        }
    }
    // We´ve reached a point where we were unable to find the actual key.
    // Lets return the nearest neighbours.
    return new Tuple<Node, Node, Node>(closestSmallerValue, null, closestBiggerValue);
}
```

Abbildung 15: Die Implementierung von GetNearestNeighbours

Abbildung 16 zeigt die Methode des Verkehrsnetzes, mit der ein Pfad registriert werden kann. Sie stellt die Registrierung von ausschließlich zulässigen Pfaden sicher und sorgt weiterhin für die Löschung alter Registrierungen durch das Eintragen jeden neuen Weges in die PathDisposalQueue. Diese wird in Abschnitt ***6.3.4*** erläutert.

```
public bool TryRegisterPath(IMovableGameObject unit, TickedPath path)
{
    if (unit == null || path == null) throw new NotImplementedException("faulty implementation");

    foreach (var tuple in path.GetTupleList())
    {
        if (!StreetGraphDict.ContainsKey(tuple.Item1)) throw new NotImplementedException("faulty implementation");
        if (!this[tuple.Item1].IsFree(tuple.Item2, tuple.Item2 + unit.Speed, unit)) return false;
    }

    // Precondition, every vertex in the path has been checked with IsFree()
    foreach (var tuple in path.GetTupleList())
    {
        if (StreetGraphDict.ContainsKey(tuple.Item1)) this[tuple.Item1].Register(tuple.Item2, unit);
    }

    // Path got registered, lets save it for later disposal from every used vertex
    AddToDisposalQueue(path);
    return true;
}
```

Abbildung 16: Pfadregistrierung im Verkehrsnetz

6.3.3 Wegsuche und Wegregistrierung

Um Einheiten die Möglichkeit zu bieten, mit Hilfe des Verkehrsnetzes von einem Startpunkt zu einem Zielpunkt zu gelangen, muss ein Wegfindungsalgorithmus implementiert werden. Weiterhin wird eine Möglichkeit benötigt, einen gefundenen Weg zu registrieren. Diese Funktionalität bietet das Verkehrsnetz an. Neben der angesprochenen TryRegisterPath (***Abbildung 16***) stellt das Verkehrsnetz noch eine **GetPath** Methode zur Verfügung, welche anhand eines Start- und Endpunktes versucht, einen Weg im Verkehrsnetz zwischen diesen beiden Punkten zu finden. Die Wegfindung wurde in dieser Arbeit nicht speziell optimiert, es kommt der Dijkstra Algorithmus [WikiDijkstra] zum Einsatz.

6.3.4 PathDisposalQueue

Jede Einheit, die aktuell einer Aufgabe nachgeht und sich auf dem Weg zu dieser befindet, braucht vom System nicht weiter beachtet zu werden. Ihr Weg ist vorberechnet und ihr Zustand muss erst wieder aktualisiert werden, wenn sie ihr Ziel erreicht hat. Aus diesem Grund gibt es im Verkehrsnetz eine PathDisposalQueue. In dieser Warteschlange werden alle Einheiten eingetragen, die sich im Verkehrsnetz befinden. Dabei werden die Einheiten mit der Endzeit ihres Weges (der Zeitpunkt, an dem sie das Verkehrsnetz wieder verlassen) in die Queue eingetragen. Das Element vorne in der Queue ist immer jenes, das als nächstes fertig wird. Die Queue erlaubt es, eine einzige Einheit in jedem GameTick zu prüfen und nicht alle im Verkehrsnetz registrierten Fahrzeuge.

Die PathDisposalQueue ist aktuell über eine einfache PriorityQueue [PriorityQ1] implementiert, welche intern über einen binary heap realisiert ist [PriorityQ2]. Die für das Projekt wichtigen Min-, Insert- und Delete-Methoden haben eine Komplexität von O(log n).

6.3.5 IdleUnitsQueue

Einheiten, die zum aktuellen Zeitpunkt keinem Auftrag nachgehen, werden vom UnitEnvironment in der IdleUnitsQueue gespeichert. Das UnitEnvironment bietet anderen Komponenten auf Anfrage Einheiten aus dieser Warteschlange an. Wird eine Einheit abgelehnt (z.B. weil ihr kein Auftrag vermittelt werden konnte), so wird sie an das Ende der Warteschlange zurückversetzt. Die Warteschlange ist zum aktuellen Zeitpunkt global für eine Spielwelt.

In einem erweiterten Stadium des Systems entfällt die IdleUnitsQueue. Einheiten sollten nicht als autonome Agenten agieren, da sie in der Regel einen Auftraggeber haben. Dies kann beispielsweise ein Gebäude sein. Denkbar ist hier eine Müllverbrennungsanlage, die Fahrzeuge aussendet, um Müll bei anderen Betrieben abzuholen. Eine Verwaltung von untätigen Einheiten findet somit nur noch indirekt in den Gebäuden statt, wenn entweder zum aktuellen Zeitpunkt keine Waren benötigt werden oder es keine Aufträge gibt, zu denen Fahrzeuge gesendet werden könnten.

6.3.6 JobCenter

Das JobCenter ist eine Klasse des JobEnvironments. Die Aufgabe des JobCenter besteht darin, noch offene unbearbeitete Jobs zu vergeben. Das JobCenter holt sich beim UnitEnvironment eine untätige Einheit und versucht, einen Job an diese Einheit zu vergeben. Je nach Konfiguration des JobCenters wird ein zufälliger Auftrag in der Welt, ein Auftrag in der Nähe der Einheit oder der dichteste Auftrag gewählt. Das JobCenter versucht anschließend einen Weg von der aktuellen Position der Einheit zur Position des Auftrags zu erhalten. Ist dies Erfolgreich wird der Einheit der Auftrag zugewiesen.

6.4 HarvestJobsTestGame

Das HarvestJobsTestGame ist eine rudimentäre Implementierung eines grafischen Clients. Dieser erlaubt es, das Spiel zu betrachten. Das TestGame verfügt über eine Kamera, von der aus die Welt dargestellt wird. Diese Kamera kann zur Laufzeit bewegt werden. Weiterhin bietet das TestGame die Möglichkeit auf Tasteneingaben zu reagieren und vordefinierte Aktionen auszuführen. Beispielhaft sei hier das schrittweise fortführen der Simulation genannt (Erhöhen des GameTick). Das TestGame lässt die Simulation kontinuierlich voranschreiten. So kann ein flüssiger Ablauf der Spielumgebung simuliert werden. Die Geschwindigkeit ist dabei änderbar und unabhängig von der Realzeit, da nur der GameTick einen Einfluss auf den Anwendungskern hat.

6.5 Probleme während der Implementierung

Im entwickelten Prototyp wurden einige Kompromisse eingegangen, um die Anwendung sowohl als Serverprogramm ohne visuelle Ausgabe, als auch als Clientprogramm mit visueller Darstellung der Geschehnisse, nutzen zu können. Um dem EnvironmentRenderer zu erlauben, nachträglich an eine Simulation angeschlossen zu werden, wurden Methoden implementiert, die das Abfragen aller Informationen innerhalb der Environments erlauben. Bei einer Weiterentwicklung der Software – und der damit verbundenen Trennung von serverseitiger und clientseitiger Software müssen diese Teile der Anwendung wieder entfernt oder in eigene Hilfs-Interfaces ausgelagert werden.

Ein weiteres Problem in der Animation von Fahrzeugen war, dass das Verkehrsnetz ein Fahrzeug nur Begutachtet, wenn es einen Auftrag vollendet oder einen neuen Auftrag antritt, da in der Zeit des Erreichens des Auftrags keine Prüfungen notwendig sind.

Zusammenfassung

Sowohl die prototypische Umsetzung der Spielwelt, als auch die des Verkehrsnetzes sind in dieser Arbeit noch sehr einfach gehalten. Der besondere Aufbau des Verkehrsnetzes sollte es aber erlauben eine sehr hohe Anzahl an Einheiten zu verwalten.

7 Test

In diesem Kapitel wird der Prototyp des Verkehrsnetzes mittels Softwaretests geprüft. Das Ziel dabei ist es, Fehler in der Implementierung oder im Konzept festzustellen. Weiterhin umfasst dieser Abschnitt der Arbeit die visuelle Testung, da hier eventuell Fehler oder ‚unschönes Verhalten' entdeckt werden kann, welches über Softwaretests nicht ohne weiteres festzustellen sind [Bat04]. Der Anspruch, eine hohe Performanz innerhalb des Verkehrsnetzes zu haben, erfordert zudem die Durchführung von Profiling [Gre09 S85 ff.] [WikiProfiling]. Hier wird vor allem Wert auf das ‚CPU Sampling' gelegt, welches rechenintensive Teilbereiche der Anwendung aufzeigt. Die Messung von ‚Memory Allocations' wird in dieser Arbeit nicht behandelt

Es wurden einige Komponententests, primär zur Aufdeckung von Berechnungsfehlern oder fehlenden Prüfungen, implementiert. Die Tests beziehen sich dabei größtenteils auf das Einfügen von Straßen, Knoten und Kanten in das Verkehrsnetz. Es wurden sonst keine Komponententests definiert. Grund hierfür ist, dass es sich bei dem Projekt um einen Prototyp handelt, bei dem die gewünschte Funktionalität auch in der Entwicklung verändert wird. Die Validierung von berechneten Fahrtwegen und der Prüfung von Zeitintervallregistrierungen an den Knotenpunkten des Straßensystems wurde über Szenariotests sichergestellt. Die Umsetzung der Tests erfolgt über das im Visual Studio vorhandene Unit Test Framework MSTest [MSTest].

Standardkonfiguration
Alle Tests gehen von folgender Grundkonfiguration aus:

- In der Software entsprechen 1000 Ticks einer echten Sekunde.
- Eine Einheit benötigt 1000 Ticks um sich um eine Distanz von eins (Die Seitenlänge eines Blocks) auf einer Achse im Koordinatensystem zu bewegen.
- Eine Einheit benötigt 100 Ticks um einen Auftrag zu bearbeiten.
- Es wird ein Auftrag pro Simulationstick vergeben.
- Die Simulation tickt in 100-Tick-Schritten.
- Die Distanz (Luftlinie) zu einem Auftrag *in der Nähe* der Einheit beträgt maximal 15 Koordinateneinheiten.
- Der Dijkstra-Suchalgorithmus [WikiDijkstra] bricht die Suche ab, wenn er mehr als die doppelte Distanz (Luftlinie) zu seinem Ziel benötigen würde.
- Ist ein Auftrag weiter als 5 Blöcke von einer Straße entfernt, so ist er nicht erreichbar.
- In der visuellen Darstellung werden 1000 Ticks pro Sekunde abgespielt.

Alle Tests wurden auf einem Windows 7 64Bit Computer mit einem AMD Phenom™ II X6 1075T 3GHz Prozessor mit sechs Prozessorkernen, 24 GB Arbeitsspecher und Grafikkarte der AMD Radeon HD 4800 Serie mit 1GB Arbeitsspeicher durchgeführt.

7.1 Szenariotests

Für die Validierung des Verkehrsnetzes wurde auf die Verwendung von Szenarios gesetzt. Ein Szenario beschreibt eine vorgefertigte Welt, welche bestimmte Bedingungen aufweist. Im Fall des Verkehrsnetzes handelt es sich um fertige Straßen, Kreuzungen und Jobs. Nach der Initialisierung des Szenarios wird der GameTick erhöht und die Simulation der Welt damit ausgeführt. In diesem Verlauf werden unterschiedliche Annahmen geprüft. Da es sich bei dem System um ein deterministisches handelt, sind bei diesem Konzept prinzipiell alle Werte manuell berechen- und validierbar. Im Folgenden wird anhand eines Beispiels die Vorgehensweise geschildert.

SzenarioFactory

Die Szenarios für den Softwaretest werden in einer speziellen SzenarioFactory gekapselt. Diese wird genutzt, um die gleichen Szenarios für die verschiedenen Testsarten (Komponententest, Systemtest, nullView-ClientTest, visueller Test) nutzen zu können. Beispielhaft wird im Folgenden ein einfaches Szenario beschrieben.

Szenario StraightSingleRoad

Dieses Szenario erstellt eine Welt, in der eine Einbahnstraße existiert. Es grenzen Aufträge an die Straße an, und es wird eine Einheit in die Welt eingefügt. In ***Abbildung 17*** ist der Methodenkopf und die Kommentarbeschreibung des Szenarios zu erkennen.

```
public static IWorld StraightSingleRoad(Cfg config, int seed, int roadLength,
                                        int jobCount, int jobOffset)
{
    // Possible Setup with StraightSingleRoadTwoUnits(conf, 1, 10, 2, 8);:
    // A = Road
    // -  0  1  2  3  4  5  6  7  8  9 10 11 X
    // 0 [>][>][>][>][>][>][>][>][>][>][>][>]
    // 1 [U]                       [J][J]
```

Abbildung 17: Szenario StraightSingleRoad

Dieses Szenario kann nun sowohl in einem Testframework, einer selbst definierten Testklasse zur Performanceanalyse (nullView-Client), als auch in einem visuellen Test verwendet werden. Der Simulationsablauf muss bei diesen Varianten, sofern sie die gleiche Konfiguration nutzen, gleich sein. In einem Testframework kann nun eine manuelle Kalkulation, wie sie in ***Abbildung 18*** zu sehen ist, zur Prüfung des Systems genutzt werden.

```
// Theoretical Path:
// 100 Ticks to find a new (first) Job (Cfg.TickStepSize)
// 1000 Ticks to get to fist Road at (0,1,0)
// 1000 Ticks per Road to get to (8,1,0) which is 8*1000 = 8.000 Ticks
// 1000 Ticks to get to the Goal (Job) at (8,1,1)
// 100 Tick to destroy goal Block
// ==============================
// 10200

// 100 Ticks to find a new (second) Job (Cfg.TickStepSize)
// 1000 Ticks to get to the Road at (8,1,0)
// 1000 Ticks per Road to get to (1,1,0) which is 1*1000 = 1.000 Ticks
// 1000 Ticks to get to the Goal (Job) at (9,1,1)
// 100 Tick to destroy goal Block
// =========
// 13.400 Ticks total
// With 10200 for first job an 3200 for second one
```

Abbildung 18: Beschreibung eines StraightSingleRoad Tests

Durch den Determinismus der Anwendung ist eine Prüfung der Simulation sehr leicht möglich. Dazu wird die Simulation an einem beliebigen Zeitpunkt angehalten und die internen Werte von Spielobjekten geprüft. Im genannten Testfall wird eine Prüfung der Wegregistrierung durchgeführt. ***Abbildung 19*** zeigt den Code des Testfalls für die Überprüfung der Zeitstempel eines Fahrzeugs. Dabei wird die Simulation soweit laufen gelassen, bis die Einheit einen neuen Auftrag annimmt. Anschließend wird die ActionQueue des Fahrzeugs, welche alle Aktionen beinhaltet, die die Einheit noch vor sich hat, begutachtet und die Zeitstempel der Aktionen mit den manuell berechneten Werten verglichen.

```
[ ... ]
// Lets tick the world once
world.Update(currentTick);
// The unit now has aquired a new job, a registration on every used vertex was made
IMovableGameObject unit = world.UnitEnv.UnitMap.GetUnit(0);
var queue = (Queue<IAction>)typeof(BasicHarvester).InvokeMember("actionQueue",
BindingFlags.GetField | BindingFlags.Instance | BindingFlags.NonPublic |
BindingFlags.Public, null, unit, null);

// Aquire the actionQueue of the unit
var actions = queue.ToArray();
// We are still where we were spawned
Assert.AreEqual(unit.Position, new Point3(0, 1, 1));
// 10 as there were initiali 11 waypoints, but first one is already the active on an not on queue
Assert.IsTrue(actions.Length == 10);
Assert.AreEqual(actions[0].ExecutionTick, 1100);  // drove to first road, drive to second
Assert.AreEqual(actions[0].EndTick, 2100); // drove to second road

Assert.AreEqual(actions[9].ExecutionTick, 10100); // drove to job, harvest
Assert.AreEqual(actions[9].EndTick, 10200); // finished job
[ ... ]
```

Abbildung 19: Prüfung der UnitQueue im Testfall

Durch die Implementierung der grafischen Oberfläche ist es zudem möglich, eine visuelle Repräsentation des hier beschriebenen Szenarios zu sehen. Der geschilderte Testfall ist in ***Abbildung 20*** im initialisierten Zustand dargestellt. Die Einheit ist dabei Gelb dargestellt. Die Einbahnstraße besteht aus zehn einzelnen Straßenstücken, die mit einander verbunden sind (es wäre auch möglich mit lediglich zwei Straßenstücken zu arbeiten). In Rot erkennt man zwei bisher unbearbeitete Aufträge, die von Einheiten angenommen werden könnten. ***Abbildung 21*** zeigt ein fortgeschrittenes Stadium der Simulation. Die Einheit hat einen Auftrag angenommen (grünes Truck-Symbol) und befindet sich bereits auf dem Weg zu diesem Auftrag. Im weiteren Verlauf der Simulation bearbeitet die Einheit den angenommenen Auftrag und nimmt nach Vollendung einen weiteren Auftrag an. Dieser Zustand ist in ***Abbildung 22*** zu sehen.

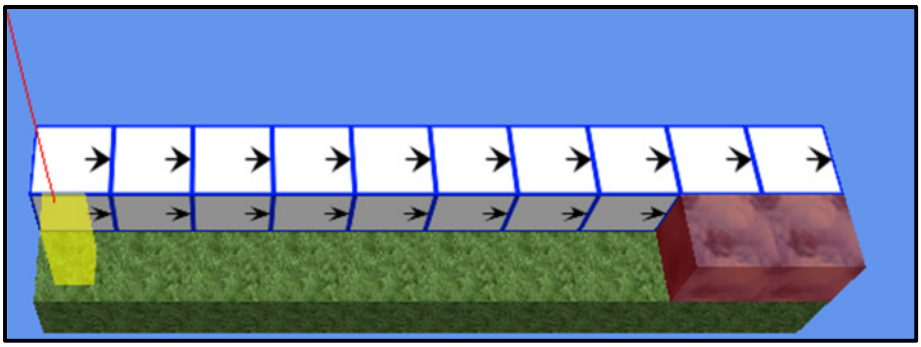

Abbildung 20: Visuelle Darstellung des Szenarios zur Initialisierung

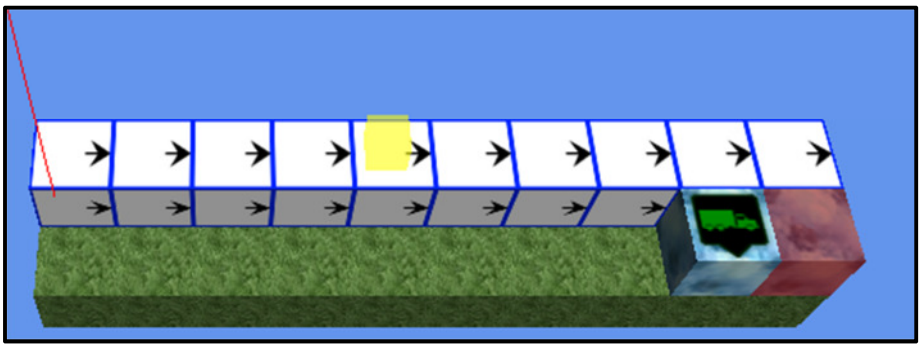

Abbildung 21: Das Fahrzeug (gelb) ist dabei einen Auftrag zu erfüllen

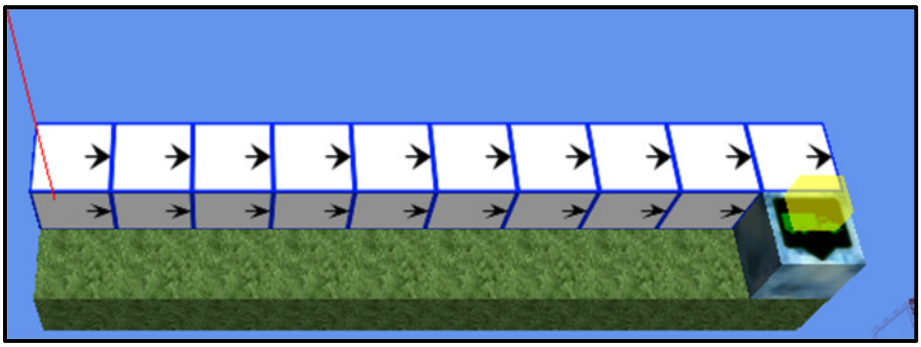

Abbildung 22: Das Fahrzeug hat einen Auftrag erfüllt und bearbeitet einen weiteren

7.2 Visueller Test

In die Spielumgebung wurde eine grafische Repräsentation eingebaut, die es erlaubt, das Verhalten der Fahrzeuge im Spiel und speziell im Verkehrsnetz zu beobachten. Die visuelle Darstellung der Szenarios ist hilfreich, um einige Spielfaktoren einschätzen zu können, die nicht, oder nur sehr umständlich, über Unittests o.Ä. abzudecken sind. Bei den Faktoren handelt es sich z.B. um den Eindruck von Geschwindigkeiten oder darum, ob das Verkehrsnetz dynamisch wirkt. In ***Abbildung 23*** erkennt man ein größeres Verkehrsnetz mit sehr vielen Einheiten, Kreuzungen und Aufträgen. Es sind nur wenige Aufträge in einem bestimmten Bereich der Spielkarte in Bearbeitung (grün-schwarze Blöcke). Die roten Striche grenzen diesen Bereich ein. Dem Algorithmus nach müssten aber überall auf der Spielkarte Aufträge erledigt werden. Im Verlauf der Simulation ist zu erkennen, dass sich die Einheiten außerhalb des rot markierten Bereichs nicht bewegen. Die Vergabe von Jobs an Einheiten ist schlicht zu langsam, und die Einheiten bearbeiten einen Job lange bevor ihnen ein neuer angeboten werden kann. Der Effekt einer zehn Mal häufigeren Jobsuche auf die Simulation, ist in ***Abbildung 24*** zu erkennen. Es existieren deutlich mehr Blöcke, die aktuell in Bearbeitung sind. Außerdem sind sie nun über das gesamte Areal verteilt. Die Fahrzeuge stehen nicht mehr einen Großteil der Zeit still und warten auf einen neuen Auftrag.

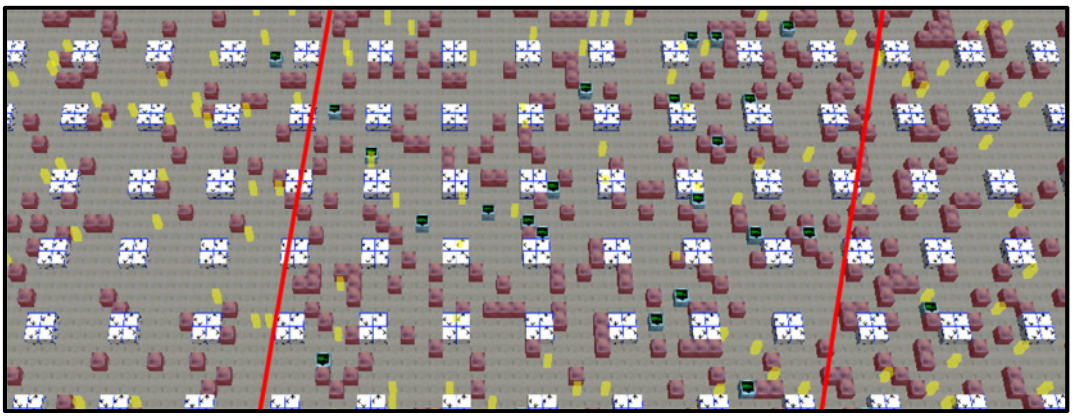

Abbildung 23: Eine Jobsuche pro 100 Ticks

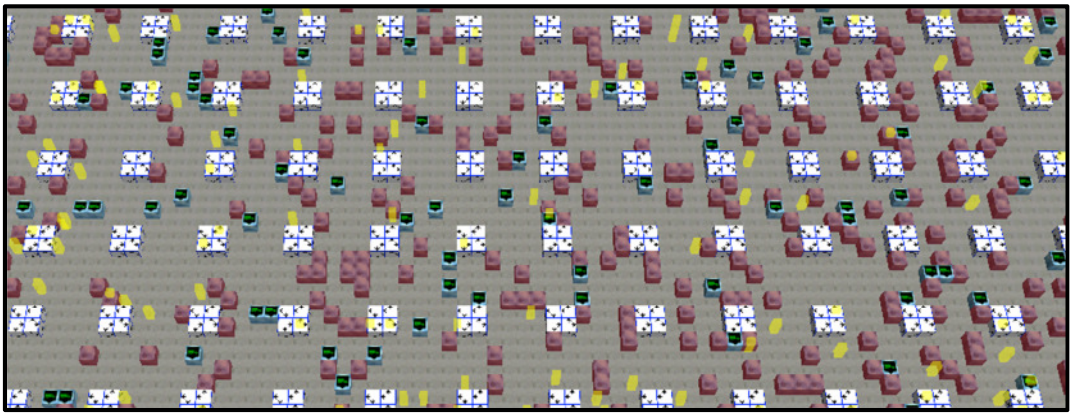

Abbildung 24: Zehn Jobsuchen pro 100 Ticks

Weiterhin wurde das visuelle Testen genutzt, um ein zu geringes Verkehrsaufkommen innerhalb des Verkehrsnetzes zu wiederlegen. Ein angedachtes Szenario zu Beginn der Arbeit war, dass nur sehr wenige Fahrzeuge gleichzeitig das Straßensystem befahren würden. Der Grund für diese Annahme war, dass die Fahrtwegregistrierung innerhalb des Verkehrsnetzes einen Großteil der offenen Aufträge anderer Einheiten blockieren würde. Im aktuellen Stadium der Arbeit ist zudem ein Zusammenstoß von Fahrzeugen möglich, wenn sie ihre Route durch das Verkehrsnetz gerade verlassen bzw. antreten. Die Kollision findet dabei technisch gesehen außerhalb des Verkehrsnetzes statt, es besteht kein Fehler in der Implementierung. Dieser Effekt könnte einem Spieler aber dennoch missfallen.

7.3 Profiling

Als Profiler [Gre09 S85 ff.] [WikiProfiling] bezeichnet man Programme, die das Laufzeitverhalten von Software analysieren. Der Anwender kann diese Analysedaten anschließend nutzen, um Problemstellen innerhalb der Software ausfindig zu machen. ***Abbildung 25*** zeigt ein Teilergebnis nach dem Durchlauf des CPU-Sampling. Diese Profiling-Methode erlaubt es, die Prozessorauslastung der getesteten Software auf verschiedene Komponenten, Klassen, Methoden oder sogar Codezeilen zurück zu führen. Das genannte Bild zeigt, dass die Wegsuche von einem Start- zu einem Zielpunkt im ausgewählten Test über 75% der Prozessorlast verursacht.

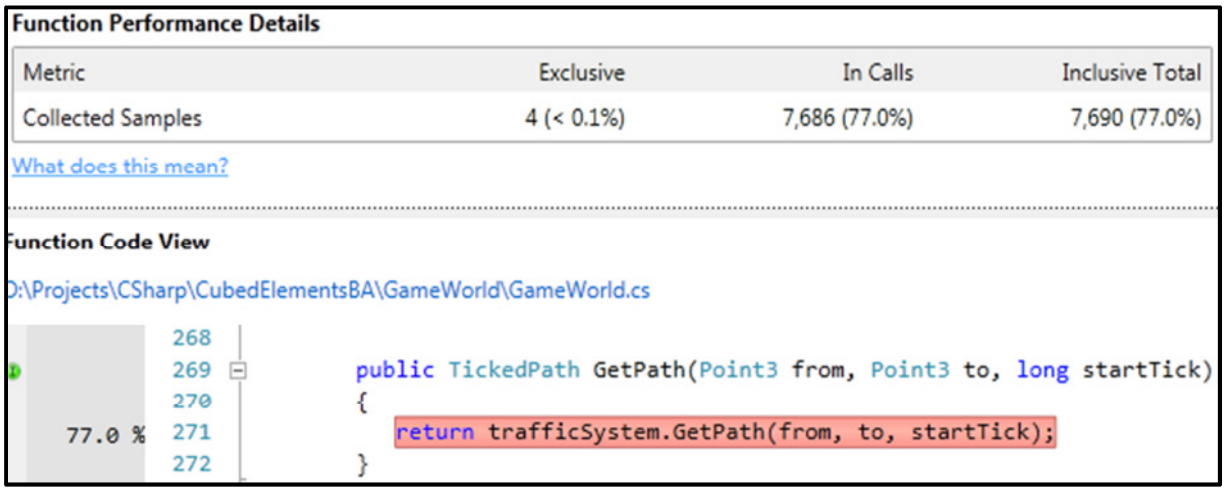

Abbildung 25: Profilerwerte für die Wegsuche

Dieses Testwerkzeug wurde während der Entwicklung der Arbeit sehr häufig verwendet und hat zu diversen Designentscheidungen geführt. Beispielhaft sind im Folgenden einige Anpassungen der Software aufgezählt:

- Die Verwendung von veränderten Red-Black-Trees (LLRB-Tree) zur Verwaltung von Zeitstempeln innerhalb der Verkehrsgraph-Knoten.

- Implementierung verschiedener konfigurierbaren Werte für die Anpassung der Wegsuche (maximale Distanz, maximale Anzahl an betrachteten Knoten, maximale Entfernung eines Jobs, maximale Anzahl an Jobsuchen pro GameTick)
- Das Halten einer „Path Disposal Queue“ innerhalb des Verkehrsnetzes, welches die Entfernung von Zeitsegmenten innerhalb der Knoten auslöst, sobald ein Fahrzeug seine Fahrt vollendet hat. Ein vorhergehender Ansatz war die regelmäßige Prüfung der Knoten auf abgelaufene Zeitsegmente. Dies führt allerdings erneut eine Grundlast in das System ein, da alle Knoten geprüft werden, egal ob es nötig ist oder nicht. Weiterhin könnten Registrierungszeitpunkte eines Fahrzeuges, das einen sehr langen Fahrweg zurücklegt, bereits teilweise entfernt sein, obwohl das Fahrzeug noch auf der Route fährt. Dies kann zu Problemen bei automatischen Tests und der Analyse von Stacktraces führen.

Weiterhin wurden durch das Profiling einige Erkenntnisse gewonnen, die noch nicht implementiert oder überprüft werden konnten:

- Die Wegsuche benötigt selbst mit Maxima für die Jobdistanz, Suchreichweite usw. etwa 50% der Prozessorzeit. Der Einsatz vom A-Stern Algorithmus [WikiAStar] könnte zu einem verbesserten Verbrauchsverhalten führen.
- Während der Laufzeit werden sehr viele Objekte erzeugt und verworfen. Beispielhaft können hier Punkte im Koordinatensystem (Point3) und Tupel genannt werden. Die Verwendung eines Objectpools könnte helfen, die Memory Allocations zu reduzieren und den Garbagecollector zu entlasten.
- Das Verkehrsnetz verwendet eine generische PriorityQueue. Es existieren performantere Varianten speziell für den Einsatz in Diskreten Event Simulationen, wie zum Beispiel die LadderQueue [TAN05], welche für Min und Insert eine Komplexität von O(1) aufweist.
- Bei jeder Auftragssuche wird derzeit geprüft, ob eine Straße in der Nähe ist. Durch den Einsatz eines Dictionarys könnte hier die jeweils dichteste Straße gecached und die Anzahl an Prüfungen stark reduziert werden.

7.4 Performanzanalyse

In diesem Abschnitt werden die Ergebnisse von zwei Testszenarios präsentiert, welche verwendet wurden, um die in den Anforderungen definierten Nicht-Funktionalen Anforderungen zu validieren. Das Ziel in den Anforderungen war, mehr als 3200 Einheiten verwalten und über 100 Aufträge pro Sekunde abarbeiten zu können. In ***Abbildung 24*** ist das erste Szenario zu sehen. Es wurden Straßen im Schachbrettmuster definiert und in dieser Struktur anhand eines Zufallswerts Aufträge und Einheiten platziert. Das Szenario wurde mit einer unterschiedlichen Spielkartengröße instanziiert. Jeder Punkt im Graph stellt die durchschnittliche Anzahl an Aufträgen pro Sekunde innerhalb eines Simulationsdurchlauf dar. Die Einheiten- und Auftragsverteilung ist dabei immer gleich geblieben. Im Szenario entsprechen 1000 Ticks in der Simulation einer Sekunde. Ist die Simulation nicht in der Lage, 1000 Ticks innerhalb einer Sekunde durch zu führen, ist sie langsamer als durch die Anforderungen gefordert. Die Simulation wird auf einer einzelnen Recheneinheit ausgeführt und so schnell wie möglich durchgeführt. Einheiten benötigen 1000 Ticks, um sich um eine Distanz von eins (Die Seitenlänge eines Blocks) im Koordinatensystem zu bewegen, sowie 100 Ticks (0,1 Sekunden), um einen Auftrag zu bearbeiten. Ist ein Auftrag 10 Blöcke entfernt, braucht die Einheit 10100 Ticks oder 10,1 Sekunden für die Erfüllung des Auftrags.

Im Anschluss folgt das zweite Testszenario, in dem das Verbrauchsverhalten des Verkehrsnetzes zur Laufzeit geprüft wird. Das Ziel ist dabei die gegen null gehende Prozessorlast durch das Verkehrsnetz selbst darzustellen.

Szenario 1.1

Das Szenario wird mit einer Auftragssuche pro 10 Ticks, das entspricht 100 Auftragssuchen pro Sekunde, gestartet. In ***Abbildung 26*** sind die Ergebnisse der Analyse in einem Graphen dargestellt. Jeder Punkt stellt einen Simulationsdurchlauf dar. Die Simulation läuft wesentlich schneller ab als benötigt. Sie ist in der Lage, über 25000 Aufträge mit einer Geschwindigkeit von fast 3000 Aufträgen pro Sekunde zu bearbeiten. ***Abbildung 27*** zeigt, dass ab etwa 5000 möglichen Aufträgen und 1500 Einheiten in der Simulation keine Vollbeschäftigung mehr möglich ist. Das bedeutet, dass die Einheiten in der Simulation teilweise still stehen, ohne einen Auftrag zu bearbeiten. Die Vergabe neuer Aufträge an Einheiten dauert somit länger als die durchschnittliche Auftragsdauer, um einen Auftrag zu erfüllen.

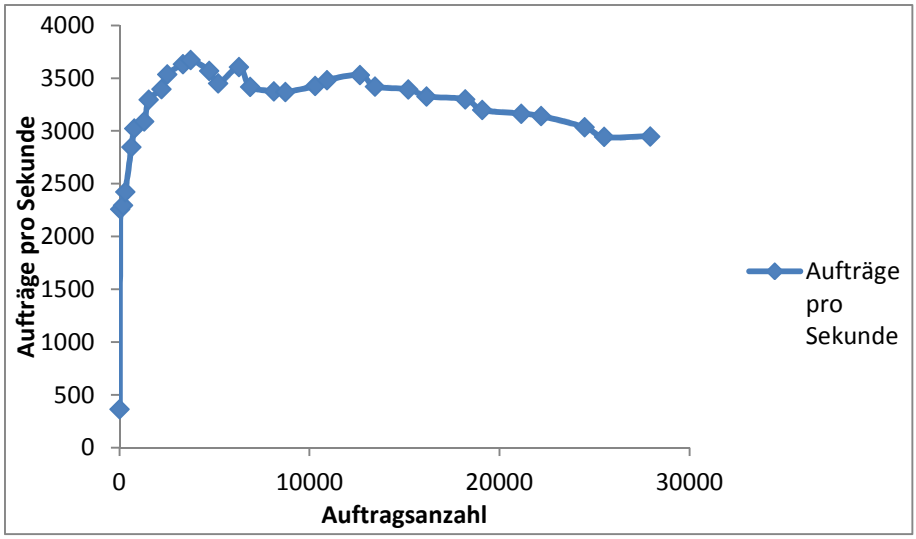

Abbildung 26: Aufträge pro Sekunde bei 100 Auftragssuchen pro Sekunde

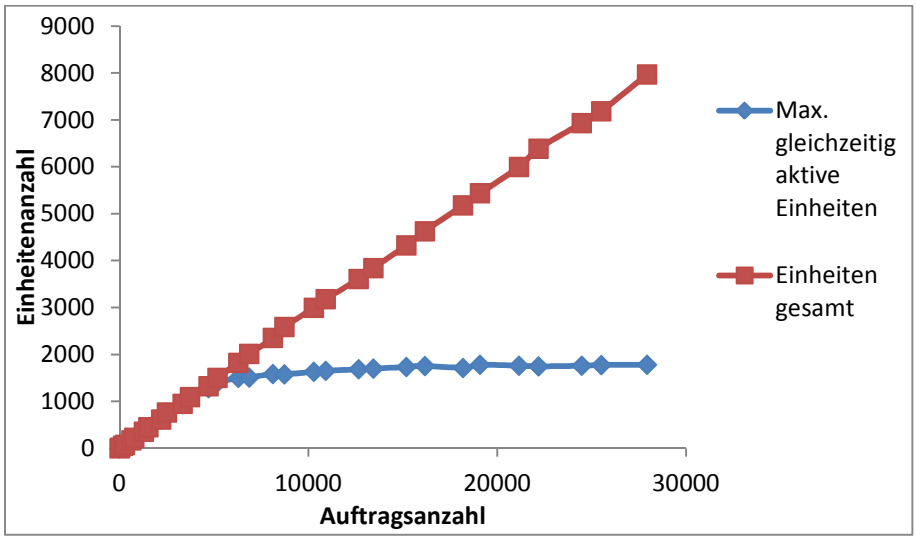

Abbildung 27: Beschäftigungsgrad bei 100 Auftragssuchen pro Sekunde

Szenario 1.2

Das Szenario wird mit einer Auftragssuche pro Tick, das entspricht 1000 Auftragssuchen pro Sekunde, gestartet. In ***Abbildung 28*** sind die Ergebnisse der Analyse in einem Graphen dargestellt. Die Simulation läuft wesentlich schneller ab, als benötigt, jedoch langsamer als in Szenario 1. Die Simulation ist in der Lage, über 25000 Aufträge mit einer Geschwindigkeit von über 2000 Aufträgen pro Sekunde zu bearbeiten. Die hohe Anzahl an Auftragssuchen erlaubt zudem eine Vollbeschäftigung der Einheiten, selbst bei etwa 8000 Einheiten in der Simulation (siehe ***Abbildung 29***).

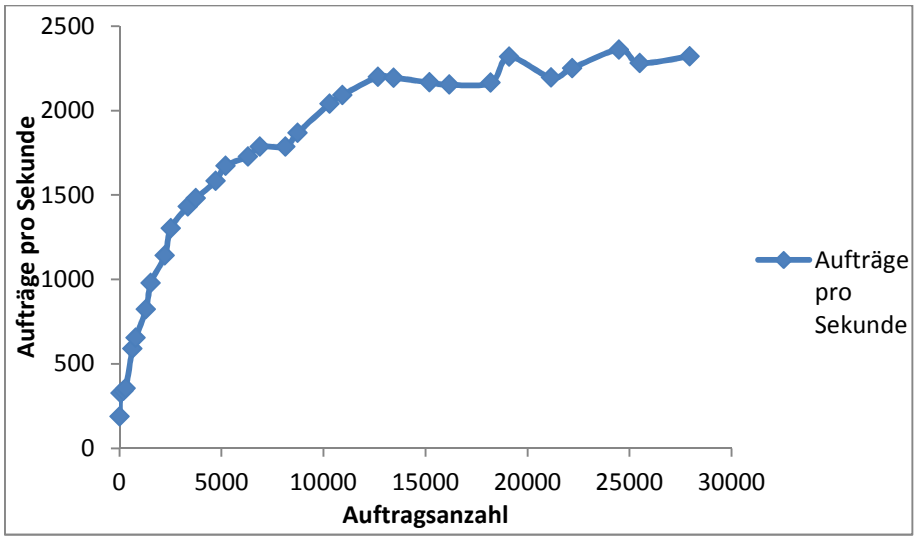

Abbildung 28: Aufträge pro Sekunde bei 1000 Auftragssuchen pro Sekunde

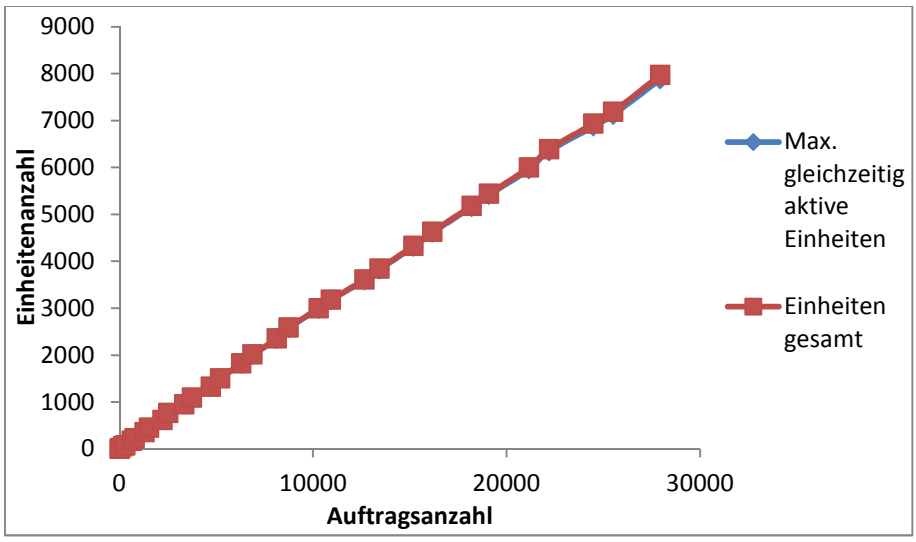

Abbildung 29: Beschäftigungsgrad bei 1000 Auftragssuchen pro Sekunde

Szenario 2.1

In diesem Szenario wird an Stelle des CPU-Sampling, welches statistische Erhebungen durchführt, das Instrumentation Profiling verwendet. Diese Methode verändert den Code, um genaue Testergebnisse liefern zu können.

Die visuelle Darstellung des Testszenarios ist in ***Abbildung 30*** zu sehen. Der Test generiert für die Anzahl an Einheiten, die dem Szenario übergeben werden, jeweils eine Straße aus neun Kreuzungen und einem Auftrag (Job) am Ende der geraden Strecke. Die Abbildung zeigt nur einen kleinen Teil des gesamten Testszenarios. Insgesamt werden 400 Einheiten in diesem Test erstellt.

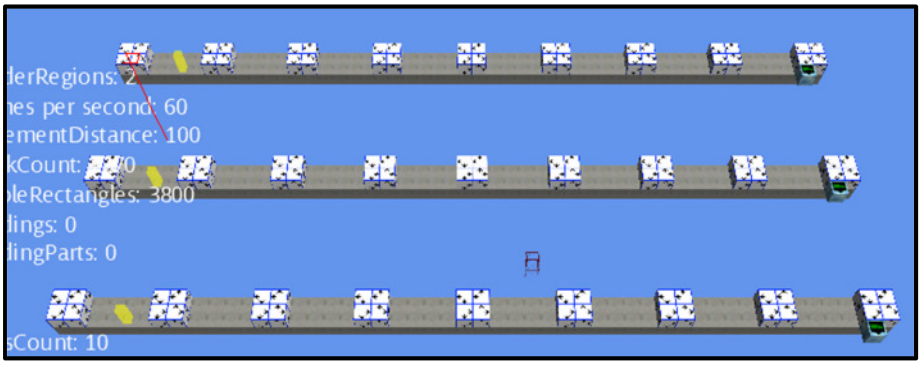

Abbildung 30: Visuelle Darstellung des IdleUnitTests

Nachdem die Welt initialisiert wurde, pausiert das Programm für einige Sekunden. Dies vereinfacht bei der Analyse der Profilingwerte die Unterscheidung von Initialisierung und Auftragssuche. Das Szenario wird so konfiguriert (siehe ***Abbildung 31***), dass alle Fahrzeuge innerhalb des ersten Tick den ihnen am dichtesten liegenden Auftrag zugewiesen bekommen. Das bedeutet, dass es keine Auftragssuchen mehr nach dem ersten Tick gibt, da alle in der Welt existierenden Aufträge bereits angenommen wurden und auch alle in der Welt existierenden Einheiten beschäftigt sind.

```
// Initialize the world with the specific values
Stopwatch s = new Stopwatch();
const int unitCount = 400;
config.TicksPerJobSearch = 100;
config.TickStepSize = 100;
config.JobSearchesPerRound = unitCount;
config.ActiveJobSearchMethod = JobSearchMethod.GetClosestJobIgnorePriority;
world = WorldScenarioFactory.MultiStraightRoad(config, 1, unitCount);
totalJobs = world.JobEnv.JobCount;
System.Threading.Thread.Sleep(2000);

s.Start();
while (!isFinished)
{
    Update(s);
}
s.Stop();
```

Abbildung 31: Initialisierung des IdleUnitTest (Szenario 2.1)

Das Profilingergebnis ist in ***Abbildung 32*** zu sehen. Der Graph zeigt die Prozessorauslastung während der Durchführung. Der Test wurde auf einem Computer mit Multicore-Prozessor (sechs Prozessorkern) ausgeführt. Im Test wird aber nur ein Prozessorkern verwendet, daher liegt die Auslastung im Graphen bei unter 20%. Die Prozessorlast ist zudem durch die Profilingtechnik erhöht. Im Folgenden wird das Ergebnis anhand der ***Abbildung 32*** erläutert.

Abschnitt 1 Initialisierung

Die hohe Auslastung zwischen null und etwa zwanzig Sekunden (A) ist die Initialisierung des Tests. Dieser Teil ist für den Test nicht relevant.

Abschnitt 2 Sleep

Der Abschnitt zwischen A und B ist das zuvor genannte Warten der Applikation. Dies dient nur der besseren Unterscheidung von Initialisierung und Auftragssuche.

Abschnitt 3 Auftragssuche

Im Abschnitt zwischen B und C findet die Auftragssuche für die 400 im Test definierten Einheiten statt.

Abschnitt 4 Fahrt

Der Gelbe Bereich im Graphen markiert die Zeit, in der alle Einheiten einem Auftrag nachgehen. Es findet keine Auftragssuche mehr statt, da Vollbeschäftigung herrscht. Unter dem Graphen ist zu sehen welche Teile der Applikation in dem Gelb markierten Bereich wie viel Zeit benötigen. Die Zeit wird fast ausschließlich mit dem Warten auf den Zeitpunkt des nächsten Simulationsschritts gefüllt. Die übrigen etwa 1.5% resultieren größtenteils aus den, in Abschnitt ***6.5*** genannten, Anpassungen des Systems, um die grafische Repräsentation zu integrieren.

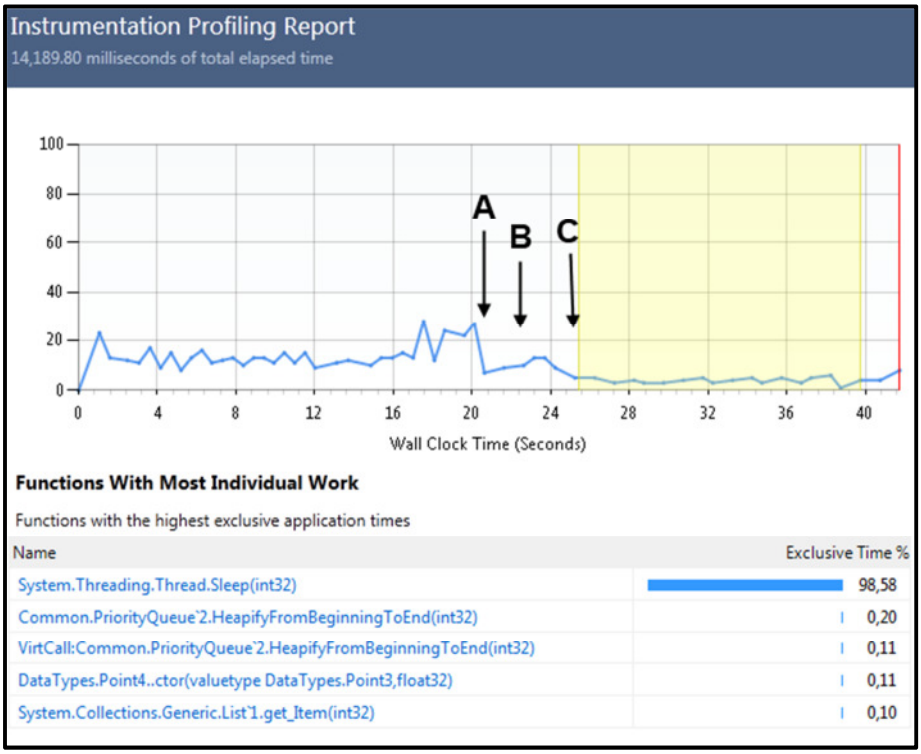

Abbildung 32: ‚Instrumentation Profiling' Ergebnis des IdleUnitTests

Szenario 2.2

Das Szenario in 2.1 wurde auch mit 8000 Einheiten und der CPU-Sampling Methode durchgeführt. Die Ergebnisse sind in ***Abbildung 33*** und ***Abbildung 34*** zu sehen. Wie auch in Szenario 2.1 sind hier die Stadien der Simulation gekennzeichnet. Bis A ist die Initialisierung. Zwischen A und B ist das Sleep (Warten zur besseren Analysierbarkeit). Zwischen B und C wird die Auftragssuche für die 8000 Einheiten ausgeführt. Ab C fahren die Fahrzeuge zu ihrem Auftrag. ***Abbildung 34*** zeigt die benötigte Rechenzeit für die verschiedenen Methoden der Applikation. Dabei nimmt die TryGetNewJob-Methode über 97% der Zeit in Anspruch. Weitere 1,5% der Zeit werden von der Update-Methode der Einheiten verbraucht. Diese ist dafür Zuständig, dass die grafische Repräsentation der Einheit aktuell bleibt und stellt die Grundlast des Systems dar. Diese soll, wie in Abschnitt ***6.5*** genannt, in einem weiteren Entwicklungsstadium wegfallen, da eine grafische Repräsentation auf dem Serversystem nicht benötigt wird.

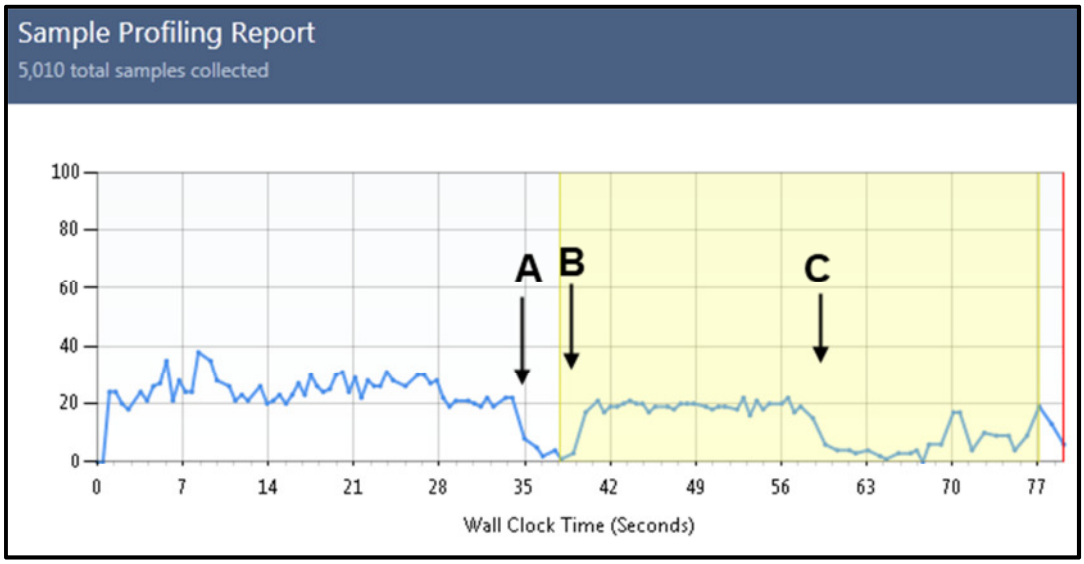

Abbildung 33: CPU-Sampling Ergebnis-Graph bei 8000 Einheiten

	Inclusive Samples %
GameWorld.Managers.Logic.UnitActionManager.Update(int64)	99,86
GameWorld.Managers.Logic.UnitActionManager.TryGetNewJob(class	97,76
GameWorld.Managers.Logic.UnitActionManager.ExecuteAction(class	1,50
GameWorld.GameObjects.Units.BasicHarvester.Update(int64)	0,70
GameWorld.Environments.UnitEnv.UnitRegion.RemoveVisualUnit(	0,36
GameWorld.Environments.UnitEnv.UnitRegion.AddVisualUnit(clas:	0,14
GameWorld.GameObjects.Units.BasicHarvester.get_Idle()	0,06
GameWorld.GameObjects.Units.BasicHarvester.get_EndPosition()	0,02
Common.PriorityQueue`2.Dequeue()	0,58
GameWorld.Environments.Managers.Graphics.UnitEnvEventManager.Upd	0,02

Abbildung 34: Zeitverbrauch der Methoden

Zusammenfassung

Bereits diese prototypische Implementation des Verkehrsnetzes ist in der Lage weitaus mehr als 100 Aufträge pro Sekunde zu bearbeiten. Mit der Vergabe von über 2000 Aufträgen pro Sekunde und fast 8000 gleichzeitig aktiven Fahrzeugen ermöglicht das System eine performante Simulation von sehr großen Verkehrsnetzen. Die Kalkulation findet dabei auf einer einzelnen Recheneinheit (Kern) statt. Multithreading wird bewusst nicht verwendet, da eine der Anforderungen darin besteht, mehrere Spielwelten performant auf einer einzelnen Recheneinheit simulieren zu können. Weiterhin zeigt Szenario 2, dass zur Laufzeit des Systems, wenn die Wegsuche vollendet ist, nahezu keine Berechnungen durchgeführt werden müssen.

8 Ergebnis

In diesem Kapitel wird das Konzept aus Kapitel *5*, die prototypische Realisierung aus Kapitel *6* und die Testergebnisse aus Kapitel *7* mit den Anforderungen aus Kapitel *4* verglichen und evaluiert. Dabei liegt der Fokus auf dem Verkehrsnetz und der Performanz.

8.1 Abgleich der Anforderungen an den Spiel-Prototyp

Im Folgenden werden alle Anforderungen aus Kapitel *4*, die den Spiel-Prototyp betreffen, aufgezählt und die Umsetzung in der Arbeit diskutiert.

Funktionale Anforderungen

Es muss eine Umgebungswelt geben, in der das Spiel stattfindet
Eine auf Voxeln [Voxel] basierende Umgebungswelt wurde in das Spiel integriert. Dieser Ansatz wurde gewählt, um bei geometrischen Berechnungen möglichst häufig mit ganzen Zahlen arbeiten zu können.

Die Umgebung sollte manipulierbar sein
Der Aufbau des Prototyps erlaubt die Manipulation der gesamten Spielwelt. Die Welt kann dynamisch vergrößert und verändert werden. Die Einheiten im Testszenario entfernen beispielsweise Bestandteile ihrer Umgebung bei der Erfüllung eines Auftrags.

Es muss eine Möglichkeit geben, Einheiten und Gebäude innerhalb der Umgebung zu erstellen
Das Erstellen von Spielobjekten, speziell das Einfügen von komplexeren aus vielen Teilen bestehenden Modellen, wurde über die AddSchematic-Methode in der GameWorld realisiert. Sie erlaubt, viele verschiedene Objekte in die Welt einzufügen. In dieser Arbeit wurden allerdings bisher keine Modelle für Gebäude definiert, da sie für das Testszenario nicht relevant waren.

Es muss Aufträge geben, die von Einheiten angenommen und abgearbeitet werden können
Das in der Implementation genannte ‚JobEnvironment' übernimmt die Verwaltung und die Vergabe von Aufträgen an Einheiten. Einheiten sind nicht als Agenten modelliert, sondern werden von der Einheitenkomponente verwaltet und bekommen Aufträge nur dann zugewiesen, wenn sie sie sicher erfüllen können.

Die Umgebung muss die Möglichkeit liefern, Straßen zu platzieren.
Straßen können genau wie Gebäude und Einheiten über die AddSchematic-Methode in die Spielwelt eingefügt werden.

Fahrzeuge müssen ausstehende Aufträge annehmen können
In diesem Punkt unterscheidet sich die Anforderung von der Umsetzung. Die Fahrzeuge sind nicht in der Lage, eigenständig Aufträge anzunehmen. Sie werden ihnen von der Einheitenkomponente (UnitEnvironment) zugewiesen. Existiert ein Auftrag, der von einer Einheit ausgeführt werden kann, so ist sie gezwungen diesen auch anzunehmen. Entscheidungsfreiheit wird im aktuellen Stadium der Arbeit nicht benötigt.

Nichtfunktionale Anforderungen

Das Spiel sollte einige Tausend Einheiten und Aufträge verwalten können
Das Ziel, mindestens 3200 Einheiten verwalten zu können, wurde erreicht. Mit etwa 8000 Einheiten (siehe Abschnitt *7.4*) wurde sogar mehr als die doppelte Menge erreicht.

Es wird großen Wert auf die Effizienz, speziell das Verbrauchsverhalten, gelegt
Das Szenario 2.1 aus Abschnitt *7.4* prüft das Verbrauchsverhalten des Prototyps, sowie das Verkehrsnetz auf das Verbrauchsverhalten, wenn keine offenen Aufträge vorliegen. Die benötigte Prozessorleistung liegt bei unter 2%. Szenario 2.2 aus Abschnitt *7.4* gibt zudem Aufschluss über die Methoden, die diese zwei Prozent ausmachen. Dabei handelt es sich größtenteils um die in Abschnitt *6.5* genannten Anpassungen, um die Anwendung visuell testbar zu machen.

Einzelne Komponenten sind individuell testbar
Die einzelnen Komponenten sind leider nicht ohne weiteres direkt testbar. Durch die eingesetzte Event-Struktur, über die sehr viele der Informationen übertragen werden, ist es nötig, der zu testenden Komponente einen Mockup der GameWorld zu übergeben, welches der Komponente die Möglichkeit bietet, sich auf verschiedene Events anzumelden. Anschließend ist jede Komponente einzeln testbar.

Das Spiel muss deterministisch sein
Die Tests haben gezeigt, dass der geforderte Determinismus vorhanden ist. Auf Grund der in Abschnitt *4.1* genannten Fließkommazahlen-Probleme wurden größtenteils ganze Zahlen bei Berechnungen verwendet. In einer Weiterentwicklung des Systems muss diesem Punkt weiterhin eine besondere Beachtung geschenkt werden.

8.2 Abgleich der Anforderungen an das Verkehrsnetz

Im Folgenden werden alle Anforderungen aus Kapitel *4*, die den Verkehrsnetz-Prototyp betreffen, aufgezählt und die Umsetzung in der Arbeit diskutiert.

Funktionale Anforderungen

Das Verkehrsnetz ermöglicht Fahrzeugen von einem Startpunkt zu einem Endpunkt im Netz zu gelangen
Die Anforderung wurde umgesetzt.

Das Verkehrsnetz ist in der Lage, eine Route von einem Start- zu einem Zielpunkt zu berechnen, wenn diese existiert
Die Anforderung wurde umgesetzt. Es gibt zum Teil Wege von einem Start- zu einem Zielpunkt, die sehr umständlich wären. Diese Wege werden aufgrund der implementierten Maximalwerte im Wege-Suchalgorithmus nicht zurück geliefert. Es ist allerdings auch möglich, das Verkehrsnetz mit sehr hohen Maxima zu starten, was dazu führt, dass selbst die genannten umständlichen Wege zurück geliefert werden und die Performanz des Systems, auf Grund der erhöhten Suchzeit, leidet. Ein denkbares Szenario, um das Verhungern von Fahrzeugen zu vermeiden, ist die Speicherung eines maximalen Entfernungswerts für die Suche innerhalb jeder Einheit. Dieser Wert könnte bei jedem fehlgeschlagenen Suchvorgang erhöht werden. Der Suchradius einer Einheit würde sich somit jedes Mal, wenn ihr kein Auftrag gegeben werden kann, erhöhen.

Das Verkehrsnetz ist zur Laufzeit erweiterbar
Die Anforderung wurde eingeschränkt umgesetzt. Die Erweiterung des Verkehrsnetzes ist möglich, es wurden aber nicht alle Fehlersituationen im Prototyp behandelt.

Das Verkehrsnetz garantiert, dass Fahrzeuge im Straßennetz nicht kollidieren
Das in Abschnitt *5.3.4* beschriebene und in Abschnitt *6.3.2* implementierte Konzept garantiert die Kollisionsfreiheit im Straßennetz. Im aktuellen Stadium des Prototyps konnten keine Fehler beim Testen aufgedeckt werden.

Fahrzeuge im Verkehrsnetz halten sich an vereinfachte Verkehrsregeln (Rechtsfahrgebot, Rechts vor Links)
Die Implementation garantiert, dass Fahrzeuge auf ihrem Weg im Verkehrsnetz weder Anhalten, noch mit andren Fahrzeugen zusammenstoßen können. Es ist daher nicht nötig, sich an Verkehrsregeln zu halten, bzw. die Regel wird indirekt erfüllt, da eine Missachtung der Regel nicht möglich ist.

Nicht-Funktionale Anforderungen

Das Verkehrsnetz sollte eine sehr hohe Anzahl an Fahrzeugen verwalten können
Wie in Abschnitt *8.1* bereits erwähnt wurde, kann das Verkehrsnetz mehr als die in den Anforderungen gewünschten 3200 Einheiten verwalten.

Das Ziel, mindestens 3200 Einheiten verwalten zu können, wurde erreicht. Mit 8000 Einheiten wurde sogar mehr als die doppelte Menge erreicht. Es wurden noch keine Tests mit mehr Einheiten durchgeführt.

Das Verkehrsnetz sollte eine gute Effizienz, speziell ein gutes Verbrauchsverhalten, haben
Die Performanz Analyse (siehe Abschnitt *7.4*) ergibt, dass das Verkehrsnetz mit mehr als 8000 Einheiten zurechtkommt. Es ist weiterhin in der Lage über 2000 Aufträge pro Sekunde an Einheiten zu verteilen. In Szenario 2.1 und Szenario 2.2 aus Abschnitt *7.4* wird zudem gezeigt, dass die Grundlast auf das System ohne neue Aufträge gegen null geht.

Das Verkehrsnetz sollte eigenständig testbar sein
Das Verkehrsnetz ist, wie in Abschnitt *8.1* dargelegt, leider nicht ohne weiteres direkt testbar. Durch die Verwendung eines Mockups der GameWorld, ist das Verkehrsnetz ohne Abhängigkeiten zu anderen Komponenten vollständig testbar.

Das Verkehrsnetz sollte ‚lebendig' auf den Betrachter wirken
Die grafische Repräsentation der Spielwelt und des Verkehrsnetzes erlaubt es, alle Spielobjekte zu beobachten. Es ist möglich, die einzelnen Fahrzeuge zu verfolgen und ihre Route zu begutachten. Auch der Stillstand von Fahrzeugen, wenn neue Aufträge zu langsam vergeben werden, ist gut zu erkennen. Bei diesen Einschätzungen handelt es sich um die des Autors.

8.3 Abgleich der weiteren Anforderungen

Folgende Punkte sind zwar mangels einer Netzwerkarchitektur nicht umgesetzt worden, der Prototyp liefert aber in einigen Punkten bereits Ansätze, die bei einer weiteren Implementation von Vorteil sind:

Sicherheit: Die Möglichkeiten zu Cheaten sollten so stark wie möglich eingeschränkt werden.
Es wurden Konzepte in den Prototyp integriert, die das Cheaten erschweren. Beispielsweise ist keine direkte Interaktion mit Objekten möglich, und die Welt ist in kleinere Teile unterteilt, die dem Spieler vorenthalten werden könnten, bis er sie benötigt. Zudem findet die Simulation vollständig auf dem Serversystem statt, der Spieler kann lediglich Maus und Tastatureingaben an den Server übertragen, die Ausführung und Validierung findet ausschließlich auf dem Serversystem statt.

Fehlertoleranz: Der Verlust von Spielobjekten bei einem Clientseitigen Absturz sollte gering sein.
Stürzt das Spiel eines Clients ab, so bleibt die Spielwelt auf dem Server intakt. Der Server erhält lediglich keine neuen Eingaben des Spielers mehr. Nach der Erkennung des Absturzes kann die Spielwelt in ihrem aktuellen Status pausiert und gespeichert werden, ohne Informationen zu verlieren.

Verbrauchsverhalten: Die Laufzeitkosten der Server-Struktur im Betrieb müssen wirtschaftlich bleiben. Ein einzelner Server sollte in der Lage sein einige Hundert Spieler zu verwalten.
Es ist im Entwicklungsstadium des Verkehrsnetzes nicht möglich über die Wirtschaftlichkeit innerhalb einer Server-Struktur eine Aussage zu treffen. Die Testergebnisse aus Kapitel ***7*** zeigen allerdings, dass das Konzept selbst bei sehr komplexen Spielwelten (siehe Abschnitt ***4.4.2***) noch eine zweistellige Spielerzahl auf einer Recheneinheit unterstützen kann.

9 Resümee

9.1 Zusammenfassung

Die Arbeit zeigt, dass es möglich ist, mehrere Tausend Fahrzeuge in einem Verkehrsnetz auf nur einem Prozessorkern zu simulieren und zu verwalten. Weiterhin ist es gelungen, die durch Kollisionsabfragen verursachte Grundlast innerhalb des Verkehrsnetzes vollständig zu beseitigen und dennoch ohne Kollisionen auszukommen. Die Software zeigt ein zufriedenstellendes Verbrauchsverhalten in Bezug auf die Prozessorzeit. Die Speicherauslastung wurde nicht analysiert. Die hier entwickelte technische Umsetzung des Straßenverkehrsnetzes lässt sich durchaus auch auf andere Verkehrsnetze übertragen, etwa die Simulation von Flug-, Wasser- oder Schienenverkehr. Dafür ist allerdings eine Behandlung von Höhenunterschieden nötig, welche in dieser Arbeit nicht berücksichtigt wurden. Durch die Modellierung des Verkehrsnetzes als Graphen sollte die Integration von Höhenunterschieden aber kein Problem darstellen. Der entwickelte Spiel-Prototyp, in welchem das Verkehrsnetz integriert ist, ist zudem leicht erweiterbar und testbar. Der Spiel-Prototyp und das Verkehrsnetz wurden so entwickelt, dass sie auf einem Serversystem ohne XNA Framework laufen können.

Problematisch an dem vorgestellten Ansatz ist vor allem die Funktionsweise des Verkehrsnetzes. Einheiten sind gezwungen, einen vorbestimmten Weg zu fahren. Dabei geht die Dynamik des Systems teilweise verloren. Schwierig ist zudem die Interaktion von Spielobjekten mit den Objekten innerhalb des Straßensystems. Unerwartete Eingriffe aus Sicht des Verkehrsflusses, wie z.B. die Explosion einer Bombe innerhalb der Spielwelt, erfordern eine besondere Behandlung. Da die Positionen aller Fahrzeuge nicht ohne weiteres berechnet werden können, kann nur auf umständliche Weise geprüft werden, ob ein Fahrzeug betroffen ist. Es müsste in diesem Fall beispielsweise jeder Verkehrsknoten in der Nähe der Explosion begutachtet und die Entfernung von passierenden Einheiten über die Zeitsegmentregistrierungen errechnet werden.

Außerdem ist die Aussagekraft der exakten Messwerte eingeschränkt, da das Verkehrsnetz für den Einsatz in MMORTS-Serversystemen gedacht ist, deren Verbrauchsverhalten, auf Grund ihrer Komplexität, nicht ohne weiteres nachzubilden ist.

Dennoch bietet dieses Konzept die Grundlage, um darauf weitere Verkehrs- und Logistiksysteme aufzubauen, welche letztendlich in eine größere Spielumgebung eingebettet werden können. Der Einsatz bleibt zudem nicht auf MMORTS beschränkt. Das Konzept ist auch für andere Spieltypen geeignet.

9.2 Ausblick

Die Software befindet sich in einem sehr frühen Stadium und erfüllt nur einen grundlegenden Teil der an ein Spiel gestellten Anforderungen. Einige naheliegende Erweiterungen, die vor allem die Performanz des Systems noch weiter erhöhen könnten, wurden in Abschnitt ***7.3*** genannt.

Das Verkehrssystem selber benötigt zwar, wie in Abschnitt ***7.4*** gezeigt, fast keine Prozessorzeit zur Laufzeit, allerdings stellt die Wegsuche noch ein großes Problem dar. Diese Verbraucht sehr viele Ressourcen auf dem Serversystem. Ein interessanter Gedanke, der diese Last nahezu vollständig von dem Serversystem entfernen könnte, wäre es, die Wegsuche auf die Clients auszulagern. Der Client würde dem Server Routen für die Einheiten anbieten, welche vom Server nur noch geprüft werden müssten. Das Cheaten ist dabei durch die benötigte Registrierung auf den Knoten nahezu unmöglich. Dem Client wäre es theoretisch möglich absichtlich schlechte Routen mit gültigen Zeitsegmenten an den Server zu liefern. Dieses Problem wäre über das vereinzelte Nachrechnen einiger Routen auf dem Server leicht zu beseitigen. Bemerkt der Server, dass ihm Wege geliefert werden, die nicht seinen eigenen Berechnungen entsprechen, kann er beispielsweise die Verbindung zum Client trennen, oder den Client als potentiellen Cheater markieren.

Bei einer Weiterentwicklung des Spielkonzepts, in dem Hunderte Spieler in einem zusammenhängenden Wirtschaftsraum Waren untereinander handeln, wird die Performanz der Clientcomputer zu einem Problem. Da in diesem Konzept auf die synchronisierte Simulation unter allen Teilnehmern gesetzt wird, kann es passieren, dass der Computer eines Teilnehmers zu langsam ist, um an der Simulation teilnehmen zu können. Dieses Problem lässt sich nur dadurch lösen, dass der Teilnehmer ausschließlich einen kleinen Teil der gesamten Welt zu einem Zeitpunkt berechnet. Dies würde aber die synchronisierte Simulation nicht mehr erlauben, da dem Teilnehmer Informationen fehlen würden. Ein Ansatz könnte hier sein, die Welt in Regionen zu unterteilen. Der Spieler erhält nur Informationen von Regionen, die sich in seiner unmittelbaren Umgebung befinden. Fahrzeuge, die über diese Regionen hinweg interagieren, müssten dem Spieler vom Server mitgeteilt werden.

Literaturverzeichnis

[Ada06] Adams, E.; Rollings, A.: Game Design and Development – Fundamentals of Game Design. Upper Saddle River, New Jersey: Pearson Education, Inc. 2006

[Ale05] Alexander, T.: Massively Multiplayer Game Development 2. Boston, Massachusetts: Charles River Media 2005

[Adams] Adams, D.: The State of the RTS. http://pc.ign.com/articles/700/700747p1.html, Abgerufen [2012-11-04]

[AgeOfEmpires] Mark Terrano, Paul Bettner: 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond. http://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php/, Abgerufen [2012-11-04]

[Arn02] Arnold, D.; Isermann, H.; Kuhn, A.; Tempelmeier, H. (Hrsg.): Handbuch Logistik. Berlin, Deutschland: Springer-Verlag 2002

[Bukkit] Bukkit: minecraft server mod API. http://bukkit.org/, Abgerufen [2012-10-31]

[CitiesXL] Focus Home Interactive: http://www2.citiesxl.com/, Abgerufen [2012-10-29]

[Cla06] Claus, V.; Schwill, A.: Duden Informatik. Mannheim, Deutschland: Dudenverlag 2006

[CounterStrike] GameData, Inc.: Tickrate / Net_Graph Overview. http://www.counter-strike.com/faqs/tickrate/, Abgerufen [2012-11-04]

[CSharp] Microsoft: CSharp. http://msdn.microsoft.com/en-us/vstudio/hh341490.aspx, Abgerufen [2012-11-04]

[Delegates] Microsoft: Delegat (C#-Referenz). http://msdn.microsoft.com/de-de/library/900fyy8e%28v=vs.100%29.aspx, Abgerufen [2012-11-04]

[Diablo32] IncGamers Ltd: Asian Diablo 3 Realm: Massive Rollback to Remove Dupes. http://diablo.incgamers.com/blog/comments/asian-diablo-3-realm-massive-rollback-to-remove-dupes, Abgerufen [2012-10-29]

[Diablo31] IncGamers Ltd: Two (Formerly) Working Duping Methods in Diablo 3. http://diablo.incgamers.com/blog/comments/two-formerly-working-duping-methods-in-diablo-3, Abgerufen [2012-10-29]

[Diablo33] Blizzard Entertainment: Auction House. http://us.battle.net/d3/en/game/guide/items/auction-house#fees, Abgerufen [2012-10-29]

[EndOfNations] Trion Worlds, Inc.: End of Nations. http://endofnations.com/de/, Abgerufen [2012-11-04]

[EndOfNationsYT] Youtube: E3 - End of Nations HD Footage/Developer Dual-commentary. http://www.youtube.com/watch?v=eN3mnVcPzr0, Abgerufen [2012-11-04]

[Eri04] Ericson, C.: Real-Time Collision Detection. San Francisco, California: Focal Press 2004

[ESA12] Entertainment Software Association: Essential Facts About The Computer And Video Game Industry. http://www.theesa.com/facts/pdfs/ESA_EF_2012.pdf, Abgerufen [2012-06-26]

[Esbensen] Esbensen, D.: GDC 2005 Proceeding: Online Game Architecture: Back-end Strategies. http://www.gamasutra.com/view/feature/2242/gdc_2005_proceeding_online_game_.php, Abgerufen [2012-10-29]

[Eve1] CCP Games: Eve Online Website. http://www.eveonline.com/de/, Abgerufen [2012-10-29]

[Eve2] Drain, B.: EVE Evolved: EVE Online's server model. http://massively.joystiq.com/2008/09/28/eve-evolved-eve-onlines-server-model/, Abgerufen [2012-10-29]

[Events] Microsoft: event (C#-Referenz).http://msdn.microsoft.com/de-de/library/8627sbea%28v=vs.100%29.aspx, Abgerufen [2012-11-04]

[Everquest] Sony Online Entertainment: Everquest. http://everquest.station.sony.com/, Abgerufen [2012-10-29]

[Fiedler] Fiedler, G.: Floating Point Determinism. http://gafferongames.com/networking-for-game-programmers/floating-point-determinism/, Abgerufen [2012-11-04]

[Gam94] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Niederlande, Amsterdam: Addison-Wesley Professional. 1994

[Gre09] Gregory, J.: Game Engine Architecture. Natick, Massachusetts: A K Peters, Ltd. 2009

[Hal08] Hall, R.; Novak, J.: Game Development Essentials: Online Game Development. Clifton Park, New York: Delmar Cengage Learning 2008

[Hat07] Hatzi, O.; Thomas, S.; Dalakas, V.; Nikolaidou, M.; Anagnostopoulos, D.: A cellular automata framework for studying expandable traffic flow models. In Proceedings of the 2007 summer computer simulation conference (SCSC). Society for Computer Simulation International 2007, Artikel 9 , 5 Seiten.

[Klausa] Marius Klausa: Konzeption und Entwicklung eines Frameworks für online basierte Aufbau- und Rollenspiele http://opus.haw-hamburg.de/volltexte/2011/1421/pdf/Bachelorarbeit_Marius_Klausa.pdf, Abgerufen [2012-11-04]

[Mat89] Mattern, F.; Mehl, H.: Diskrete Simulation – Prinzipien und Probleme der Effizienzsteigerung durch Parallelisierung. Informatik-Spektrum (1989) 12, 198–210

[Minecraft] Mojang: Minecraft. https://minecraft.net/, Abgerufen [2012-10-31]

[Nag99] NAGEL, K.; SCHRECKENBERG, M.: A cellular automaton model for freeway traffic. Journal de Physique I 2, 12 (1992) 12, 2221–2229.

[OpenTTD] OpenTTD Website. http://www.openttd.org/en/, Abgerufen [2012-10-31]

[Planetside] Sony Online Entertainent: Planetside 2. http://www.planetside2.com/, Abgerufen [2012-11-04]

[RuneScape] Wikia, Inc.: RuneScape clock. http://runescape.wikia.com/wiki/RuneScape_clock, Abgerufen [2012-11-04]

[Sal03] Salen, K.; Zimmerman, E.: Rules of Play – Game Design Fundamentals. London, England: The MIT Press 2004

[SecondLife] Linden Research, Inc.: SecondLife. http://secondlife.com/, Abgerufen [2012-10-29]

[Siedler] Ubisoft: Siedler Website. http://siedler.de.ubi.com/siedler-7/, Abgerufen [2012-10-29]

[SimCity5] EA Games: SimCity. hhttp://www.simcity.com/en_US, Abgerufen [2012-10-30]

[SimCity5YT] Youtube: Maxis Live Broadcast: SimCity. http://www.youtube.com/watch?v=6vqg_kBqPBg&feature=plcp, Abgerufen [2012-10-31]

[SimCityGlassBox] EA Games: Sim City Glassbox Engine. http://www.simcity.com/en_US/game/info/glassbox-engine, Abgerufen [2012-10-31]

[Smith1] Smith, F.: Synchronous RTS Engines and a Tale of Desyncs. http://www.altdevblogaday.com/2011/07/09/synchronous-rts-engines-and-a-tale-of-desyncs/, Abgerufen [2012-11-04]

[Smith2] Smith F.: Synchronous RTS Engines 2: Sync Harder. http://www.altdevblogaday.com/2011/07/24/synchronous-rts-engines-2-sync-harder/, Abgerufen [2012-11-04]

[StacklessPython] Stackless: Stackless Python. http://www.stackless.com/, Abgerufen [2012-10-31]

[Starcraft2] Stardepot: Ticks per second, waits, timers, real time, game time. http://www.stardepot.org/50/ticks-per-second-waits-timers-real-time-game-time, Abgerufen [2012-11-04]

[SUPER12] SuperData Research, Inc.: Global MMO Games Spending Exceeds $12 Billion. http://www.superdataresearch.com/global-mmo-games-spending-exceeds-12bn/, Abgerufen [2012-06-26]

[TransportTycoon] Wikipedia: Transport Tycoon. http://de.wikipedia.org/wiki/Transport_Tycoon, Abgerufen [2012-10-31]

[Travian] Travian Games GmbH: Travian. http://www.travian.de/, Abgerufen [2012-10-29]

[VisualStudio] Microsoft: Visual Studio 2010. http://www.microsoft.com/germany/VisualStudio, Abgerufen [2012-11-04]

[WikiAnno] Wikipedia: Anno (Spieleserie). http://de.wikipedia.org/wiki/Anno_%28Spieleserie%29, Abgerufen [2012-10-30]

[WikiAStar] Wikipedia: A*-Algorithmus. http://de.wikipedia.org/wiki/A*-Algorithmus, Abgerufen [2012-11-08]

[WikiCitiesXL] Wikipedia: Cities XL. http://en.wikipedia.org/wiki/Cities_XL, Abgerufen [2012-10-30]

[WikiComputersimulation] Wikipedia: Computersimulation. http://de.wikipedia.org/wiki/Computersimulation, Abgerufen [2012-10-29]

[WikiDijkstra] Wikipedia: Dijkstra-Algorithmus. http://de.wikipedia.org/wiki/Dijkstra-Algorithmus, Abgerufen [2012-11-08]

[WikiFlow] Wikipedia: Maximum flow problem. http://en.wikipedia.org/wiki/Maximum_flow_problem, Abgerufen [2012-11-08]

[WikiFrame] Wikipedia: Frame rate. http://en.wikipedia.org/wiki/Frame_rate, Abgerufen [2012-10-29]

[WikiFrame2] Wikipedia: Bildfrequenz. http://de.wikipedia.org/wiki/Bildfrequenz, Abgerufen [2012-10-29]

[WikiHappyFarm] Wikipedia: Happy Farm. http://en.wikipedia.org/wiki/Happy_Farm, Abgerufen [2012-11-04]

[WikiIEEE754] Wikipedia: IEEE floating point. http://en.wikipedia.org/wiki/IEEE_floating_point, Abgerufen [2012-10-29]

[WikiMinecraft] Wikipedia: http://de.wikipedia.org/wiki/Minecraft#Rezeption, Abgerufen [2012-10-31]

[WikiMMO] Wikipedia: Massively multiplayer online game. http://en.wikipedia.org/wiki/Massively_multiplayer_online_game, Abgerufen [2012-10-29]

[WikiMMOList] Wikipedia: List of massively multiplayer online games. http://en.wikipedia.org/wiki/List_of_massively_multiplayer_online_games, Abgerufen [2012-10-29]

[WikiONotation] Wikipedia: Big O notation. http://en.wikipedia.org/wiki/Big_O_notation, Abgerufen [2012-10-29]

[WikiProfiling] Wikipedia: Profiling (computer programming). http://en.wikipedia.org/wiki/Profiling_%28computer_programming%29, Abgerufen [2012-11-08]

[WikiRTS] Wikipedia: Real-Time Strategy. http://en.wikipedia.org/wiki/Real-time_strategy, Abgerufen [2012-11-04]

[WikiRTSList] Wikipedia: List of MMOGs. http://en.wikipedia.org/wiki/List_of_MMOGs#Real-time_strategy, Abgerufen [2012-11-04]

[WikiSimCity] Wikipedia: SimCity. http://de.wikipedia.org/wiki/SimCity, Abgerufen [2012-10-30]

[WikiSupCom] Wikipedia: Supreme Commander. http://de.wikipedia.org/wiki/Supreme_Commander, Abgerufen [2012-11-09]

[WikiXNA] Wikipedia: Microsoft XNA. http://en.wikipedia.org/wiki/Microsoft_XNA, Abgerufen [2012-11-04]

[Wilkes] Wilkes, I.: Second Life's Architecture. http://www.infoq.com/presentations/Second-Life-Ian-Wilkes, Abgerufen [2012-10-29]

[XNA] Microsoft: XNA. http://msdn.microsoft.com/en-us/aa937791.aspx, Abgerufen [2012-11-04]

Glossar

Bottleneck: Flaschenhals; Eine Engstelle im System, welche die Performanz maßgeblich beeinflusst.

CPU Sampling: Stichprobenartige Analyse der Prozessorauslastung.

Deadline: Spätester Zeitpunkt, an dem etwas fertig gestellt sein muss.

Delegate: Delegat; Ein Funktionszeiger in C#, mit dem beispielsweise Methoden als Parameter übergeben werden können.

Determinismus: Die Vorbestimmtheit eines Vorgangs (z.B. ein Algorithmus). Das Ergebnis ist durch die klar definierte Arbeitsweise vorbestimmt.

Diskrete Event Simulation: Ereignisorientierte Simulation; Die Simulation wird nur durch Ereignisse fortgeführt.

Engine: Motor/Antrieb, in der Spieleentwicklung gibt es Spiel-Engines, Grafik-Engines, Physik-Engines usw. Die Engine bezeichnet dabei den Teil der Software, der die Kalkulation in einem Bereich übernimmt.

Event: Ereignis; Bei Eintreten eines bestimmten Ereignisses werden alle Zuhörer informiert. Siehe Observer-Pattern.

Forschung: Eine Forschung bezeichnet in Computerspielen i.d.R. die Erforschung von besseren Technologien, die einen Vorteil im Spielverlauf bringen.

Frame: Einzelbild; Ein Spiel läuft mit einer gewissen Anzahl an Frames pro Sekunde ab.

Lag: Verzögerung; Ist die Netzwerkverbindung gestört oder langsam, so kann es zu Verzögerungen in der Informationsübertragung kommen.

Memory Allocations: Arbeitsspeicher vergabe; Im Zusammenhang mit Profilern bezeichnet dies die Analyse von Arbeitsspeicher-Platzzuweisungen.

Mockup: Attrappe; Ein Softwareteil, welcher grundlegend so agiert wie die noch nicht fertiggestellte Software, die er vertritt.

nullView-Client: Bezeichnet nach [Ale05] ein Programm, das erlaubt, ein Spiel auch ohne grafische Repräsentation zu starten und zu testen.

GameTick: Siehe Tick.

HLSL: High Level Shader Language, Programmiersprache für die Ansteuerung der Grafikkarte über DirectX.

Instrumentation Profiling: Bezeichnet eine Profiling-Methode, bei der das zu testende Programm mit weiterem Code ausgestattet wird, um sehr genaue Testwerte zu erhalten.

Objectpool: Menge an vorinitialisierten Objekten, die von Komponenten genutzt und zurückgegeben und wiederverwendet werden können. Reduziert die Zahl an Allokationen.

Observer-Pattern: Beobachter Entwurfsmuster; Erlaubt die Weitergabe von Änderungen an einem Objekt an andere Objekte, die von dieser Änderung beeinflusst werden.

Open-World-Spiel: Ein Spiel, in dem der Spieler eine große Spielwelt geboten wird, die auch komplett zugänglich und teilweise änderbar ist.

Persistenz: Speicherung von Daten in nicht-flüchtigen Speichermedien.

Resimulation: Als Resimulation wird in dieser Arbeit das nachberechnen einer Simulation bezeichnet.

Rendering: Berechnen; Bezeichnet in der Computergrafik die Erzeugung eines Bildes aus Rohdaten.

Tick: Ein Tick bezeichnet eine Zeiteinheit.

Tickrate: Die Frequenz in der Nebenrechnungen in einem Spiel vorgenommen werden.

Verhungern: In dieser Arbeit bezeichnet Verhungern die lange oder andauernde Inaktivität eines Fahrzeugs.

Voxel: Kombination aus ‚volumetric' und ‚pixel'; Bezeichnet einen Datenpunkt in einem dreidimensionalen Raum, der ein bestimmtes Volumen einnimmt.

Zellulärer Automat: Modellierungsmethode für diskrete dynamische Systeme.

Abbildungsverzeichnis

A Anhang

Wertetabellen zur Performanzanalyse

One Job per 10 Ticks				
Aufträge	Aufträge pro Sekunde	Max. gleichzeitig aktive Einheiten	Einheiten gesamt	Durchführungsdauer in Millisekunden
15	365,8537	2	2	41
52	2260,87	12	12	23
193	2297,619	54	54	84
308	2425,197	76	76	127
635	2847,534	175	175	223
795	3022,814	227	227	263
1304	3090,047	358	358	422
1540	3297,645	452	452	467
2221	3396,024	616	619	654
2519	3537,921	762	763	712
3346	3633,008	942	948	921
3748	3674,51	1081	1090	1020
4724	3570,673	1284	1324	1323
5198	3451,527	1404	1505	1506
6297	3606,529	1503	1826	1746
6878	3418,489	1512	2011	2012
8135	3376,92	1585	2353	2409
8740	3369,314	1576	2585	2594
10299	3426,148	1632	2998	3006
10923	3481,989	1651	3182	3137
12668	3529,674	1685	3609	3589
13447	3422,499	1699	3842	3929
15198	3394,684	1733	4329	4477
16171	3328,736	1756	4628	4858
18198	3299,728	1717	5182	5515
19101	3199,498	1779	5439	5970
21157	3163,427	1759	5998	6688
22200	3141,361	1746	6391	7067
24488	3035,201	1760	6931	8068
25510	2944,028	1776	7185	8665
27943	2949,752	1779	7968	9473

Testfall Szenario 1.1 Wertetabelle

One Job per Tick				
Aufträge	Aufträge pro Sekunde	Max. gleichzeitig aktive Einheiten	Einheiten gesamt	Durchführungsdauer in Millisekunden
15	187,5	2	2	80
52	327,044	12	12	159
193	339,7887	54	54	568
308	355,248	76	76	867
635	589,0538	175	175	1078
795	654,321	227	227	1215
1304	823,7524	358	358	1583
1540	978,399	451	452	1574
2221	1140,729	618	619	1947
2519	1303,156	763	763	1933
3346	1431,138	948	948	2338
3748	1480,838	1088	1090	2531
4724	1583,11	1322	1324	2984
5198	1672,458	1502	1505	3108
6297	1727,572	1820	1826	3645
6878	1786,03	2009	2011	3851
8135	1785,949	2346	2353	4555
8740	1867,921	2583	2585	4679
10299	2039,406	2992	2998	5050
10923	2090,526	3175	3182	5225
12668	2199,306	3600	3609	5760
13447	2193,996	3829	3842	6129
15198	2165,883	4315	4329	7017
16171	2152,689	4614	4628	7512
18198	2164,367	5159	5182	8408
19101	2317,52	5418	5439	8242
21157	2194,709	5962	5998	9640
22200	2250,836	6350	6391	9863
24488	2360,289	6878	6931	10375
25510	2281,141	7128	7185	11183
27943	2321,233	7881	7968	12038

Testfall Szenario 1.2 Wertetabelle

B Inhalt der DVD

Auf der beigelegten DVD befinden sich in folgende Dateien:

Bachelor.pdf	Die Bachelorarbeit als PDF-Datei
CubedElementsBA	Der Sorucecode des Prototyps (Ordner)
Reports	Die verwendeten Profiler-Ergebnisse in Kapitel *7*

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____