der Bundeswehr
Universität München

Visualisierung von multisensorischen Daten zur Vorbereitung von Datenfusion am Beispiel einer militärischen Lage

Bachelor-Arbeit von
Stephan Tzschoppe
1006374

UniBwM – IB 16/2009

Aufgabenstellung:
Prof. Dr. Stefan Pickl

Betreuung:
Dipl.-Inform. Marco Schuler

Institut für Theoretische Informatik,
Mathematik und Operations Research
Fakultät für Informatik
Universität der Bundeswehr München

Neubiberg
31.12.2009

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken und Zitate sind als solche kenntlich gemacht.

Es wurden keine anderen, als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt. Die Arbeit wurde weder einer anderen Prüfungsbehörde vorgelegt, noch veröffentlicht.

Neubiberg, 31. Dezember 2009 _____

Unterschrift

Inhaltsverzeichnis

Eidesstattliche Erklärung **3**

1 Einleitung **7**
1.1 Motivation 7
1.2 Ziel der Arbeit 8
1.3 Aufbau der Arbeit 8

2 Verwendete Technologien **9**
2.1 Java 9
2.2 XML 10
2.3 jMonkeyEngine 11
2.4 Google Code 11

3 Theoretische Grundlagen (Multisensorische Daten und ihre Visualisierung) **13**
3.1 Multisensor-Daten 13
3.1.1 Eigenschaften von Multisensor-Daten 13
3.1.2 Fusion als notwendige Voraussetzung zur Datenverarbeitung . . . 13
3.2 Visualisierungsmöglichkeiten 13

4 Entwurf und Umsetzung einer Visualisierungsumgebung **15**
4.1 Entwurf eines Visualisierungsframeworks 15
4.1.1 Eingabedaten und Datenmodell 16
4.1.2 Übersicht über das Frontend des Frameworks 22
4.1.3 Übersicht über das Backend des Frameworks 24
4.1.4 Zusammenspiel der Komponenten 29
4.2 Visualisierung einer militärischen Lage 33
4.2.1 Problemdarstellung 33
4.2.2 Lösungsansatz 35
4.2.3 Umsetzung mithilfe des Visualisierungsframeworks 35

4.3 Kernaspekte der Implementierung 42
4.3.1 Observer-Pattern bei abgeleiteten Klassen 42
4.3.2 Entfernungsberechnung auf der Erde 43
4.3.3 Informationsverlust durch Projektion einer Kugelkappe auf eine Ebene 46
4.3.4 Mousepicking 48
4.3.5 Bewegungskegel 49

5 Fazit und Ausblick **53**
5.1 Bewertung 53
5.2 Weiterführende Arbeit 53
5.3 Fazit 53

Literaturverzeichnis **54**

Abbildungsverzeichnis **56**

Listingverzeichnis **58**

1 Einleitung

1.1 Motivation

Wir leben in einer Zeit stetig voranschreitender Technologisierung. Dies äußert sich für jeden sichtbar auf vielerlei Art und Weise. Massenspeicher mit vor Jahren noch unvorstellbaren Kapazitäten, stetig wachsende Prozessorleistung und massiver Fortschritt in der Datenübertragung sind hier nur als Beispiele zu nennen. Solche Entwicklungen sind es, die das Sammeln, Verarbeiten und Speichern von riesigen Datenmengen erst ermöglichen. Diesen Fortschritt gilt es zu nutzen und auf mögliche Anwendungsfelder auszuweiten.

Betrachtet man ein Schlachtfeld, sei es im Rahmen einer Übung, einer kriegerischen Auseinandersetzung oder eines Konflikts, so werden auch hier Informationen gesammelt und ausgewertet. Man stelle sich folgende Situation vor: Drei eigene Panzer bewegen sich durch das Gelände. Plötzlich klären sie zwei feindliche Fahrzeuge auf. Jeder einzelne eigene Panzer setzt eine Meldung ab und beschreibt, was er sieht. Dies führt zu sechs Meldungen. Der S2[1]-Offizier muss nun aus diesen Meldungen ein Lagebild erstellen. Dabei sind aus der vorhandenen (teilweise redundanten) Information zwei statt sechs feindliche Fahrzeuge zu extrahieren.

Für das geschilderte Beispiel scheint es nicht notwendig, diese Informationsauswertung zu automatisieren. In der Realität hingegen erfordert es viel Zeit, diese Arbeit zu erledigen[2]. Und genau dies ist ein großes Problem, denn je älter die Lageinformation ist, umso weniger aussagekräftig ist sie. Entscheidungen, die darauf basierend getroffen werden, können falsch oder unverhältnismäßig sein. Um diesen Missstand zu beseitigen, gilt es, den S2-Offizier bei seiner Arbeit technisch zu unterstützen.

[1] Der S2-Offizier ist verantwortlich für die militärische Sicherheit, militärisches Nachrichtenwesen mit Aufklärung und Zielfindung, elektronisch Kampfführung und eben die Wehrlage

[2] Ein Panzerbataillon der Bundeswehr umfasst zum Beispiel ca. 40 Kampfpanzer

1.2 Ziel der Arbeit

Eine technische Unterstützung kann in unterschiedlichen Abstufungen geschehen. So können die separaten Meldungen zu einer großen Meldung zusammengefasst werden. Dies ist aber nicht sonderlich hilfreich, wenn zum Beispiel aus vielen einzelnen Datensätzen einfach eine zusammenhängende Datensammlung erstellt wird. Denn wenn diese in einem textuellen Format vorliegt, ist sie von einem Menschen nur schwer zu verstehen. Weiterhin können die auftretenden Redundanzen nicht einfach erfasst, geschweige denn überhaupt genutzt werden.

Aus diesem Grund möchte ich mich in dieser Arbeit mit Visualisierungsformen auseinandersetzen, die eine menschenlesbare Sicht auf multisensorische Daten bieten. Zuerst sollen Visualisierungsmöglichkeiten aufgezeigt werden. Bei unterschiedlichen Ansätzen sollen diese bewertet werden.

Ergebnis dieser Betrachtungen wird ein Framework zur Darstellung multisensorischer Daten. Dieses verwende ich dann prototypisch dazu, die eingangs erwähnte Problemstellung des S2-Offiziers zu bearbeiten und ihm eine, für seine Lageerstellung hilfreiche, Sicht auf die eingehenden Meldungen zu geben.

Die Erweiterbarkeit des Frameworks soll durch dessen Verwendung zur Darstellung von NDP [ABK] Daten gezeigt werden.

1.3 Aufbau der Arbeit

2 Verwendete Technologien

In diesem Kapitel werden die wichtigsten Technologien, die im Laufe der Arbeit verwendet werden, vorgestellt und erläutert.

2.1 Java

Am Anfang der Arbeit stand die Wahl der Programmiersprache, die für die Implementierung verwendet werden sollte. Java in der Version 1.6 bot sich aus mehren Gründen an.

Auf der einen Seite sprach ein ganz pragmatischer Grund dafür: Da Java im Rahmen der Vorlesung über Objektorientierte Programmierung gelehrt wurde, entfiel die Einarbeitungszeit. Auf der anderen Seite gibt es aber weitere Gründe für die Wahl. An die prototypische Implementierung wurde die Anforderung der Plattformunabhängigkeit gestellt. Dies spricht für die Wahl einer Sprache, die auf einer virtuellen Maschine basiert. Im Gegensatz zu einem Java-Programm müsste ein in C++ geschriebenes auf einem anderen Rechner neu kompiliert werden. Dies erhöht den Installationsaufwand und ist keine echte Unabhängigkeit.

Unter den Sprachen, die auf einer Virtuellen Maschine basieren fällt aus Gründen der Plattformunabhängigkeit die .Net-Familie aus, da es sich hier um Microsoft- und somit betriebssystemspezifische Sprachen handelt. Weitere Gründe für die Wahl von Java waren existierende Klassenbibliotheken für die Verwendung von XML (siehe 2.2) und die Arbeit mit OpenGL als Programmierschnittstelle für die dreidimensional Visualisierung.

Ein Nachteil von Java ist die Umsetzung von GUI-Elementen. Zwar gibt es mit Swing und SWT leistungsfähige Bibliotheken, aber das Layout der grafischen Benutzeroberfläche ansprechend zu gestalten, stellt sich damit als schwierig heraus. Für die Umsetzung

der grafischen Benutzeroberfläche wurde Swing gewählt, da es Bestandteil der mit Java ausgelieferten Klassenbibliothek ist

Weiterhin ist die Verwendung von Java für die Arbeit mit 3D-Grafik umstritten (siehe [Dav05]). Ein großer Kritikpunkt ist dabei der Geschwindigkeitsverlust, der sich aus dem Interpretieren des Programms durch die Virtuelle Maschine ergibt. Wie stark dieser ins Gewicht fällt, ist umstritten. Da Geschwindigkeit für diese Arbeit aber nicht Ausschlaggebend ist, kann dieser Kritikpunkt vernachlässigt werden.

2.2 XML

Die Daten für das Testen der prototypischen Entwicklung stammen von einem Gefechtssimulator. Sie sind im XML-Format abgespeichert. Um mit diesen Daten in Java arbeiten zu können, müssen sie in Java-Klassen überführt werden. An dieser Stelle kamen zwei Möglichkeiten in Frage: Auf der einen Seite hätte nur die prototypische Implementierung, die mit den Daten arbeitet, diese eingelesen. Auf der anderen Seite konnte die Funktionalität, aus XML-Dateien zu laden, auch in das entwickelte Framework integriert werden.

Weil sich XML für einen plattform- und implementierungsunabhängigen Datenaustausch eignet und heute zu diesem Zweck ein Quasistandart ist [CITE]. Fiel die Entscheidung, das Laden von Daten im XML-Format im Framework zu implementieren. Die gewählte Programmiersprache eignete sich für diesen Zweck besonders gut, da bereits fertige Parser-Bibliotheken mitgeliefert werden.

Hier standen DOM und SAX zur Auswahl. Beide haben Vor- und Nachteile. Welche Implementierung gewählt wird, hängt vom zu lösenden Problem ab. Wird mit großen Datenmengen gearbeitet und müssen diese oft neu geladen werden, ist SAX im Vorteil, weil es sich dabei um eine sehr performante Umsetzung handelt. Ein großer Nachteil ist die eingeschränkte Traversierbarkeit der Daten. Das hängt mit der Arbeitsweise des Parsers zusammen. Er arbeitet ereignisgesteuert. Das bedeutet, die XML-Datei wird sequentiell gelesen. Wird beispielsweise ein Element gefunden, wird eine Methode zur Verarbeitung von Elementen aufgerufen. Somit kann bei der Verarbeitung kein Rückschluss auf die Position eines Elements in der XML-Datei geschlossen werden. Genauso wenig kann auf Vorgänger, Nachfolger unter Unterelemente explizit zugegriffen werden.

Der große Vorteil des DOM-Parsers ist eben die bei SAX nicht vorhandene Traversierbarkeit. Der Parser erzeugt beim Einlesen der XML-Datei einen Baum. Für jedes strukturelle Element der Daten wird ein Knoten erzeugt. Ein XML-Element ist ein Beispiel für einen solchen Knoten, aber auch ein Attribut. Auf diese Art und Weise ist es zum einen möglich, sich komfortabel durch die Daten zu bewegen. Weiterhin ist es so möglich, das Lesen von XML-Dateien in das erstellte Framework zu integrieren und für jede Implementierung mit lediglich geringem Implementierungsaufwand eine Leseroutine umzusetzen. Das Framework ist dann in der Verantwortung die Elemente zu extrahieren, die einen Datensatz darstellen. Der Anwender des Frameworks bekommt dann die Knoten der Datensätze und liest eigenständig die von ihm benötigten Informationen aus. Aus diesem Grund wurde DOM für diese Arbeit ausgewählt.

Die Tatsache, dass DOM nicht so sehr für große Datenmengen geeignet ist, weil das Erzeugen der Baumstruktur vor allem speicherplatzintensiv ist. Dieser Nachteil wird aber zum Wohle des einfacheren Umgangs in Kauf genommen.

2.3 jMonkeyEngine

2.4 Google Code

Bei der Entwicklung des praktischen Anteils dieser Arbeit bestand die Notwendigkeit, die erstellten Daten sicher abzulegen. Dafür eignet sich unter anderem die Arbeit auf einem Terminal-Server, wir er von der Universität der Bundeswehr angeboten wird. Dies ermöglicht, von unterschiedlichen Orten und an unterschiedlichen Rechnern zu arbeiten. Nachteil dieser Lösung ist jedoch, dass ein Speichern unterschiedlicher Versionen des erstellten Programms und auch dieser Arbeit nur schwer möglich ist.

Aus diesem Grund wurde für das im Rahmen dieser Arbeit erstellte Programm, aber auch für die schriftliche Ausarbeitung, ein Versionierungssystem verwendet. Aus der Lehre bereits bekannt war Subversion. Es wurde auch hier wegen der nicht mehr notwendigen Einarbeitung gewählt.

Google stellt auf seinem Portal GoogleCode die Möglichkeit bereit, eine Projekt mit Hilfe von Subversion zu versionieren. Dieser sogenannte *Project Hosting Service* hat zudem noch weitere Vorteile als nur eine komfortable Datensicherung: Der Projektstand kann

online dokumentiert werden. Somit wird es möglich, den Betreuer dieser Arbeit auf dem neusten Stand zu halten. Er kann über das Abonnieren eines RSS-Feeds jederzeit Änderungen Nachverfolgen. Weiterhin bietet Google ein Wiki an, was die Vorstellung und die Dokumentation des Programms ermöglicht. Zusätzlich gibt es eine Downloadverwaltung in der Testversionen Screenshots und ähnliches zur Verfügung gestellt werden können.

Wie oben angesprochen, wurde nicht nur der praktische Teil der Arbeit, also der Quellcode, über das Versionierungssystem gesichert. Auch die schriftliche Ausarbeitung in Form des Rohzustands (TEX-Dateien) und des fertigen PDF-Kompilats wurde auf im Googleprojekt gespeichert.

Zusammenfassend sind die Vorteile des oben geschilderten Vorgehens, dass der Quellcode und die schriftliche Ausarbeitung zuverlässig gesichert werden konnten. Der Zugriff auf diese war ortsunabhängig über das Internet möglich. Mit Hilfe der Versionierung konnte nach Experimenten auf stabile Versionen zurückgegriffen werden. Durch RSS-Feeds war es dem Betreuer jederzeit möglich, den aktuellen Stand der Arbeit zu betrachten und die im Downloadbereich befindliche Version zu testen.

3 Theoretische Grundlagen (Multisensorische Daten und ihre Visualisierung)

3.1 Multisensor-Daten

3.1.1 Eigenschaften von Multisensor-Daten

3.1.2 Fusion als notwendige Voraussetzung zur Datenverarbeitung

[Var97] [HL97] [Hal92]

3.2 Visualisierungsmöglichkeiten

4 Entwurf und Umsetzung einer Visualisierungsumgebung

In diesem Kapitel werden der Entwurf eines Visualisierungsframeworks und eine prototypische Implementierung auf dessen Basis beschrieben. Dazu werden die einzelnen Komponenten des Frameworks erklärt und ihr Zusammenspiel erläutert. Basierend auf dem Wissen über die Bestandteile können die Möglichkeiten der Erweiterbarkeit und der Individualisierbarkeit skizziert werde. Die Grundlage hierfür stellt zum einen eine prototypische Implementierung einer Visualisierung einer militärischen Lage und des Weiteren der Versuch, Daten des NDP [ABK] dreidimensional darzustellen.

Abschließend werden interessante Aspekte der Implementierung aufgezeigt, die neben den umgesetzten Implementierungsentscheidungen auch andere Wege aufzeigen sollen.

Dem Leser, der sich vorrangig für den entstandenen Prototyp interessiert, sei der Abschnitt 4.2 empfohlen. Die Details des Frameworks sollten für das Verständnis des Funktionsumfangs und die Bedienung keine Rolle spielen, können aber bei Bedarf nachgeschlagen werden.

4.1 Entwurf eines Visualisierungsframeworks

Das Ergebnis dieser Arbeit soll nicht nur eine potentielle Visualisierungs-Umgebung sein, sondern ein Framework. Es soll zum einen die Möglichkeit bietet, mit geringem Aufwand Daten anzeigen zu können, auf der anderen Seite aber umfangreiche Erweiterungsmöglichkeiten zulassen. Wie das Framework konkret entworfen und umgesetzt wurde, soll im Folgenden dargestellt werden.

4.1.1 Eingabedaten und Datenmodell

Grundlage für die weitere Arbeit sollen Eingabedaten sein, die bestimmte Voraussetzungen erfüllen. Diese gestellten Bedingungen werden im Folgenden kurz beschrieben. Darauf basierend wird das Datenmodell erläutert, in das die Quelldaten überführt werden sollen.

Eingabedaten

Um Daten dreidimensional visualisieren zu können, müssen diese bestimmte Voraussetzungen erfüllen. Diese sollen hier kurz aufgezeigt werden.

Identifizierbarkeit Unabhängig von der Art der Visualisierung ist es unabdingbar, dass jedes einzelne Datum identifizierbar ist. Aus diesem Grund sollte in den Quelldaten bereits eine eindeutige Benennung vorliegen. Sollte das nicht der Fall sein, muss dieser Misstand spätestens beim Importieren der Daten behoben werden.

Probleme, die sich ergeben können, wenn diese Bedingung nicht beachtet wird, sind zum einen, dass keine aussagekräftigen Berechnungen auf den importierten Daten durchgeführt werden können. Auch ist die Markierung eines gezielten Datensatzes unmöglich.

Lokalisierbarkeit Aus dem Ziel, die Eingabedaten im dreidimensionalen, kartesischen Raum darzustellen, ergibt sich eine ganz logische Voraussetzung: Es sollte mindestens für jede der drei Dimensionen eine Eigenschaft der Daten existieren, die eine Positionierbarkeit möglich macht. Zwar ist es genauso möglich, die Daten auf einer Linie anzuordnen und somit nur eine Positionierungseigenschaft vorauszusetzen oder sich analog auf den zweidimensionalen Raum zu beschränken. Der daraus resultierende Informationsverlust muss aber in Kauf genommen werden.

Die Voraussetzungen an den Typ, der für die Lokalisierung herangezogenen Eigenschaften, sind nicht sehr streng. Es ist zielführend, wenn es sich hier um kontinuierliche oder zumindest diskret ganzzahlige Größen handelt. Ist dies nicht der Fall, so müssen diese lediglich zur Berechnung der Anzeigekoordinaten mit einer geeigneten Abbildungsvorschrift umgerechnet werden.

mittelbar geeignet	*unmittelbar geeignet*
IP [ABK] Adressen	Entfernungsangaben
MAC [ABK] Adressen	Ortsangaben (Länge/Breite)
Zeichenketten allgemein	Zeitangaben

Tabelle 4.1: Beispiele für unmittelbar und mittelbar geeignete Größen

Weitere Eigenschaften Die Darstellung von Daten im dreidimensionalen Raum, lediglich basierend auf der Position und dem Namen, bietet noch keine hohe Anschaulichkeit. Um die in [Verweis, Visualisierungsmöglichkeiten] dargestellten Visualisierungen auch verwenden zu können, braucht es weitere Eigenschaften, die in optische Information umgewandelt werden kann.

Die Art der Daten, die für solche zusätzliche Veranschaulichung verwendet werden kann, muss nicht genau spezifiziert werden. Trotzdem gibt es Typen, die sich für bestimmte Visualisierungsmöglichkeiten besser eignen. Als einfaches Bespiel seien hier Aufzählungsdatentypen genannt, die sich geradezu anbieten, die Objekte in unterschiedlichen Farben darzustellen. Dabei können auch unterschiedliche Enumerationen gleichzeitig visualisiert werden, indem pro Eigenschaft eine Farbe verwendet wird, die je nach Wert der Eigenschaft in unterschiedlichen Helligkeiten dargestellt wird. Der Kreativität sind hier keine Grenzen gesetzt.

Datenmodell

Das im Rahmen dieser Arbeit entwickelte Datenmodell setzt den Grundstock für das später entworfene Framework. Es hat zum Ziel, durch generische Gestaltung in der Lage zu sein, unterschiedlichste Quelldaten ohne Anpassung speichern zu können.

Umgesetzt wurden diese Anforderungen durch die Trennung in eine Datensammlung und deren Daten auf der einen und Eigenschaften auf der anderen Seite.

Datensammlung mit Daten Wie in Abbildung 4.1 dargestellt, besteht das wesentliche Datenmodell aus zwei Klassen.

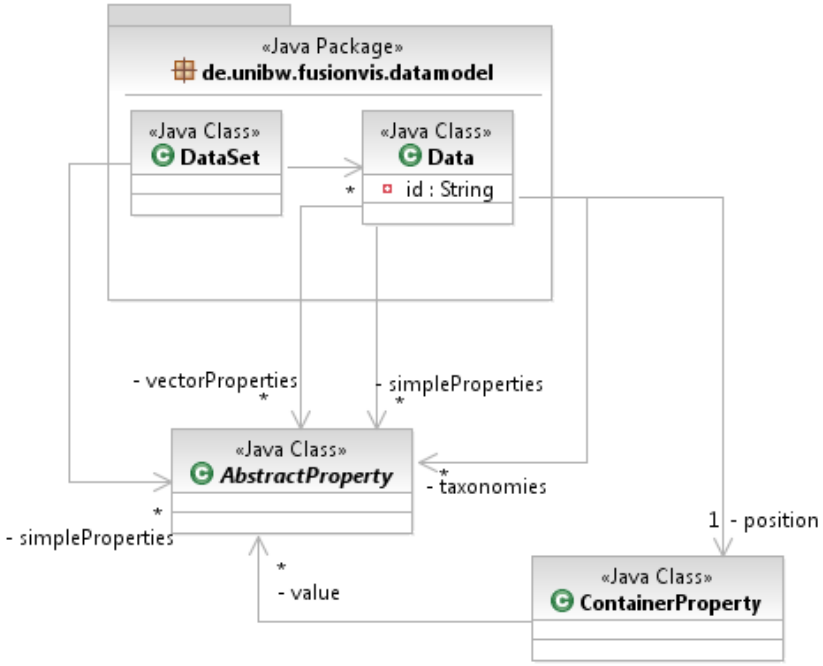

Abbildung 4.1: Übersicht des Datenmodells

DataSet Die Klasse *DataSet* kapselt die einzelnen Datensätze in Form einer Liste. Diese Klasse ist die zentrale Datenstruktur, die für alle weiteren Prozesse die notwendigen Informationen bereithält. Sie besteht im Wesentlichen aus zwei Bestandteilen. Der erste ist die angesprochene Liste der gespeicherten Daten. Weiterhin kann sie Eigenschaften erfassen, die nicht einem bestimmten Datum zu eigen sind, sondern global für die gesamte Datensammlung gelten. Das Benutzen dieser Eigenschaften ist aber fakultativ, um eine hohe Flexibilität zu gewährleisten. Die Typisierung der Eigenschaften wird weiter unten dargestellt.

Data Die Klasse *Data* kapselt ein einzelnes Datum. Wie bereits in Abschnitt 4.1.1 dargelegt, sind die notwendigen Bestandteile eines Datensatzes ein Bezeichner, der innerhalb einer Datensammlung eindeutig sein muss, sowie eine Positionsangabe. Diese ist mithilfe einer zusammengesetzten Eigenschaft festgehalten. Das Eigenschaftssystem wird weiter unten noch detailliert beschrieben.

Zusätzlich zu diesen beiden obligatorischen Angaben erfasst die Data-Klasse weiterhin drei Listen von Eigenschaften.

- einfache Eigenschaften
- zusammengesetzte Eigenschaften

- Taxonomien

Die einfachen Eigenschaften sind in der Lage textuelle und numerische Merkmale eines Datensatzes zu erfassen. Sie sind Grundlage für die spätere Visualisierung. Die zusammengesetzten Eigenschaften ermöglichen das Ablegen von vektoriellen Größen, wie zum Beispiel einer Blickrichtung, Geschwindigkeiten oder Beschleunigungen usw. Auch ist es möglich, mit ihrer Hilfe baumartig strukturierte Eigenschaften zu erfassen.

Die Taxonomien umfassen eine Liste möglicher Klassifikationen, die einem Datensatz zu eigen sein können. Auch ihre Angabe ist nicht zwingend erforderlich.

Der Aufbau einer *Data*-Klasse kann in Abbildung 4.2 nachvollzogen werden.

Eigenschaften Die Eigenschaften der Datensammlung, wie auch die der Daten an sich, sollten so generisch aufgebaut sein, dass sie für möglichst viele unterschiedliche Anwendungsgebiete ohne Änderung und Anpassung übernommen werden können.

Beim Entwurf fiel auf, dass die auftretenden Eigenschaften nach zwei Kriterien zerfallen. Auf der einen Seite kann nach Dimension der Eigenschaften unterschieden werden. So kann zum Beispiel gespeichert werden, ob die Einheit in einem Schlachtfeldsimulator

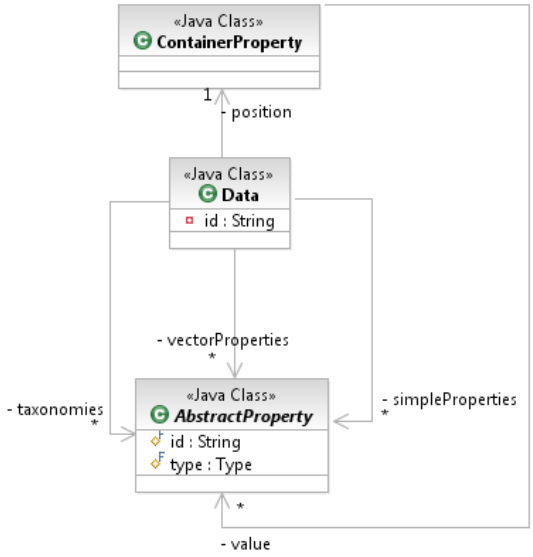

Abbildung 4.2: Aufbau der Klasse Data

feindlich, freundlich oder neutral ist. Weiterhin wäre es möglich, eine Gewichtsangabe zu speichern. In beiden Fällen handelt es sich um eine einfache, weil eindimensionale, Eigenschaft.

Im Gegensatz dazu gibt es zusammengesetzte Eigenschaften wie vektorielle Größen (Ausrichtung, Beschleunigung). In diesen Bereich fallen auch Gliederungsinformationen oder Unterstellungsverhältnisse. Eine zusammengesetzte Eigenschaft besteht damit entweder aus einfachen oder wiederum aus zusammengesetzten Eigenschaften. Diese Unterscheidung führte im Entwurf zur Auswahl des Composite-Patterns (nach [GHJV02]) für die Modellierung des Sachverhalts.

Das zweite Kriterium, in das die Informationen der Daten zerfallen, ist ihr Typ. Um einen Kompromiss zwischen einem kompakten Datenmodell und einem breiten Spektrum unterstützter Typen zu gewährleisten, fiel die Entscheidung auf folgende Datentypen:

- Boolean
- Char
- Date [ANM]
- Float
- Integer
- String

Um die grundlegenden Bedürfnisse zu stillen, hätte auch die Auswahl eines Fließkommadatentyps, mit dem sich auch ganze Zahlen darstellen lassen, und ein Zeichenkettendatentyp, mit dem alle anderen Informationen gespeichert werden können, ausgereicht. Eine noch kleinere Teilmenge, die nur den String-Datentyp umfasst, wäre unter Ausnutzung von programmiersprachenspezifischen Typumwandlungen auch denkbar gewesen. Diese beiden Möglichkeiten wurden aber im Hinblick auf die komfortablere Handhabung verworfen.

Abbildung 4.3: Eigenschaftssystem im Datenmodell

Der resultierende Entwurf ist in Abbildung 4.3 dargestellt. Er besteht aus der abstrakten Eigenschaft, den konkreten Implementierungen nach Datentypen und der Zusammengesetzten Eigenschaft.

AbstractProperty Die abstrakte Klasse dient als Muster für die konkreten Implementierungen. Sie wird über einen Bezeichner eindeutig identifiziert. Es bietet sich hier an, menschenlesbare Namen zu verwenden, die zugleich beschreibenden Charakter haben. Zusätzlich hat jede Eigenschaft einen Typ. Dieser ist bereits in der abstrakten Klasse implementiert, um zur Laufzeit ohne Kenntnis der genauen Implementierung den jeweiligen Typ der Eigenschaft abfragen zu können.

Der Satz an abstrakten Methoden bildet die notwendige Schnittstelle, um den Wert einer Eigenschaft zu lesen und zu schreiben. Hier stellt sich die Frage, warum eine abstrakte Klasse und nicht etwa ein Interface gewählt wurde. Die Antwort ergibt sich aus der Möglichkeit, die Typinformationen und den Bezeichner in der allgemeinen Eigenschaft vorhalten zu können. Das entlastet den Programmierer bei der Umsetzung der konkreten Implementierungen, da er sich darum nicht mehr kümmern muss.

Aus Sicht des Composite-Patterns stellt *AbstractProperty* die Component-Klasse dar.

Typ-Property Die Implementierungen der abstrakten Klasse tragen in der Umsetzung die Verantwortung, einen zum Typ passenden Wert zu speichern. Dieser wird in der Ausgestaltung der Manipulationsmethoden gesetzt und gelesen. Wo notwendig sind Typumwandlungen durchzuführen, bei unsinnigen Operationen (zum Beispiel das Umwandeln eines bool'schen in ein Datum) sollten Exceptions geworfen werden.

Eine Typ-Property ist die im Composite-Pattern als Leaf bezeichnete Klasse.

ContainerProperty Die zusammengesetzte Eigenschaft verwaltet ihre zugehörigen Eigenschaften in einer Liste von *AbstractProperty*-Klassen. Sie kann aus diesem Grund aus Typ-Property-Klassen bestehen oder wiederum aus zusammengesetzten Eigenschaften. *ContainerProperty* stellt im Pattern den Composite dar.

Erweiterbarkeit Das Datenmodell ist an der Stelle der Eigenschaften erweiterbar. Um einen neuen Typ einzuführen müssen vier Schritte durchgeführt werden:

In der Enumeration der Typen ist der neue einzuführen. Weiterhin müssen in der *AbstractProperty* die Zugriffs- und Manipulationsmethoden für den neuen Typ abstrakt definiert werden. Daraus folgt, dass bestehende Typen diese Methoden implementieren müssen. Macht eine Type Casting Sinn, dann kann diese umgesetzt werden. Meist können die Methoden jedoch auf das Werfen einer Exception reduziert werden.

Abschließend muss nun eine neue Typ-Property Klasse erstellt und implementiert werden, die von ihrem abstrakten Vorbild erbt. Durch diese vier Schritte hat man somit das Datenmodell um einen Typ erweitert.

4.1.2 Übersicht über das Frontend des Frameworks

Das erstellte Framework besteht im Wesentlichen aus zwei Teilen. Auf der einen Seite stehen die für den Benutzer sichtbaren Klassen, an denen nicht notwendigerweise eine Anpassung vorgenommen werden muss, um ein lauffähiges Programm zu erstellen. Zu diesen Klassen gehört auf oberster Ebene die *FusionVisForm*. Das ist die, dem Benutzer angezeigte, Programmoberfläche. Sie besteht aus einem Menü mit implementierter Funktion zum Auswählen von XML-Dateien und zum Schließen des Programms.

Die Form enthält weiterhin zwei Panels, die es dem Benutzer ermöglichen, eine textuelle und eine visuelle Darstellung der Daten zu sehen.

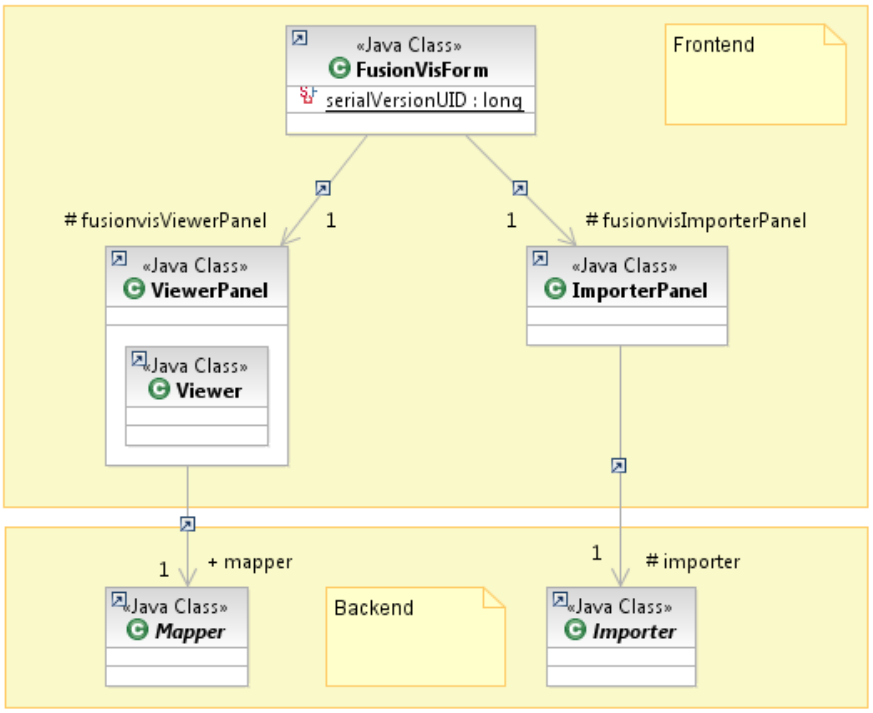

Abbildung 4.4: Übersicht des Frameworks (Frontend/Backend)

ImporterPanel Für die textuelle Sicht ist das ImporterPanel vorgesehen. Es bietet die Möglichkeit, die Daten nach bestimmten Eigenschaften zu filtern und diese mit ihren Eigenschaften in Textform anzuzeigen. Das Model, auf dem gearbeitet wird, ist das in Abschnitt 4.1.1 beschriebene Datenmodell. Somit ist das ImporterPanel auch völlig unabhängig von dem genauen Inhalt der Daten, solange diese in das beschriebene Model überführt wurden. Dafür ist der Importer zuständig, der im nächsten Abschnitt beschrieben wird.

ViewerPanel Das zweite Panel ist das ViewerPanel. Es ist verantwortlich für die dreidimensionale Anzeige der Daten. Dazu ist es notwendig, das vorhandene Datenmodell in die Struktur eines Szenenbaums zu überführen. Der Szenenbaum wird in [REF Kapitel 4 jme] beschrieben. Dieser Schritt wird vom *Mapper* bewerkstelligt, der im nächsten Abschnitt erklärt wird.

Der erstellte Szenenbaum kann nun von einem Viewer angezeigt werden. Dieser beinhaltet eine OpenGL-Anzeigefläche, welche die dreidimensionale Darstellung übernimmt und es dem Benutzer erlaubt, sich in den visualisierten Daten zu bewegen, Datensätze mittels Mousepickings (siehe Abschnitt 4.3.4) auszuwählen

und diese somit im *ImporterPanel* zu inspizieren. Auch der umgekehrte Weg ist möglich, also das Auswählen eines Datums in der textuellen Ansicht, was eine Hervorhebung seiner Visualisierung im ViewerPanel zur Folge hat.

Zusammenfassend kann man zum Frontend des Frameworks sagen, dass die von den konkreten XML-Daten unabhängigen Funktionen bereits implementiert sind. Der Anwender muss lediglich festlegen, was wie dargestellt werden soll. Das *Was* ist der Prozess des Überführens von XML in das Datenmodell des Frameworks und wird im *Importer* festgeschrieben. Das *Wie* ist die Frage nach der Art und Weise, wie die Daten dreidimensional dargestellt werden sollen und wird im *Mapper* beantwortet.

4.1.3 Übersicht über das Backend des Frameworks

Das Backend besteht im Wesentlichen aus den zwei oben erwähnten Komponenten, *Importer* und *Mapper*. Im Folgenden soll die Funktion der beiden erläutert und notwendige Schritte der Spezialisierung beschrieben werden.

Die Spezialisierung bezieht sich auf das Implementieren abstrakter Methoden, die in den beiden als abstrakt gekennzeichneten Klassen die eigentliche Funktion enthalten sollen.

Importer

Der *Importer* ist, wie bereits im vorherigen Abschnitt angemerkt, dafür verantwortlich, die Daten aus einer XML-Datei zu lesen. An die Struktur muss eine Voraussetzung gemacht werden, um den Import automatisiert durchführen zu können: Die Datensätze müssen als Elemente des ersten Elements unter dem Wurzelknoten in der XML-Dateien stehen. Ein Beispiel für eine XML-Datei, die diese Anforderung erfüllt, ist in Listing 4.1.

Ist die Voraussetzung erfüllt, liest der *Importer* die XML-Datei aus und extrahiert die DOM-Knoten der Datenelemente. Die Wahl des DOM-Parsers für diese Aufgabe wird noch in [REF] genauer erläutert.

```
1 <Situation>
    <Units>
3    <Unit>
```

```
      <Name>FriendlyTank1</Name>
      <Location>
        <Lat>52.796629714678467</Lat>
        <Lon>9.89990561649954</Lon>
        <LastModified>2009-02-20T10:25:36+01:00</LastModified>
      </Location>
    </Unit>
    <Unit>
      <Name>FriendlyTank2</Name>
      <Location>
        <Lat>52.794038961891268</Lat>
        <Lon>9.9011727699025922</Lon>
        <LastModified>2009-02-20T10:25:36+01:00</LastModified>
      </Location>
    </Unit>
  </Units>
</Situation>
```

Listing 4.1: Beispiel für eine wohlgeformte XML-Datei

Der *Importer* hält bereits notwendige Datenstrukturen vor. Zum einen sind dies zwei Membervariablen (*id*, *position*). Diese dienen dem Speichern der XML-Elementnamen, die den Wert für Position und Bezeichner eines Datums liefern. Ähnlich gibt es drei Listen, die die Elementnamen enthalten, die als einfache, zusammengesetzte Eigenschaften oder als Taxonomie extrahiert werden.

Spezialisierung Zur Spezialisierung eines *Importers* für ein bestimmtes Datenformat müssen zwei Schritte durchlaufen werden. Als erstes sollten die in das Datenmodell als Eigenschaft von Daten zu übernehmenden XML-Elemente auf die oben angesprochenen Membervariablen und Listen verteilt werden.

Als nächstes muss nun die Methode *extractDataFromNode()* implementiert werden. Ihr wird ein Knoten des DOM-Dokumentbaums übergeben. Dieser zeigt auf ein Element, dass als Datum in das Datenmodell übernommen werden soll. Die Aufgabe der Funktion ist es nun, nach den Vorstellungen des Anwenders ein Data-Objekt zu instanzieren, das die in den vorher spezifizierten Listen stehenden Eigenschaften enthält.

Hilfsfunktionen In der *Importer*-Klasse sind bereits zwei Hilfsfunktionen implementiert, die dem Anwender die Arbeit beim erzeugen der Data-Objekte vereinfachen sollen.

Die Methode *parseDate()* dient dem Auslesen eines Datums aus einem String. Für die genauere Dokumentation sei hier auf das JavaDoc verwiesen.

Die Methode *extractSimpleProperty(Node, Type)* erzeugt eine einfache Eigenschaft (siehe 4.1.1) eines angegebenen Typs aus einem DOM-Knoten. Um beispielsweise in Zeile 4 von Listing 4.1 eine Eigenschaft *Name* als String zu extrahieren, muss die Hilfsfunktion lediglich mit der Node aufgerufen werden, die auf dieses Element zeigt. Zusätzlich wird der gewünschte Typ (hier: *Type.TString*) übergeben.

```
  protected Data extractDataFromNode(Node unitNode) throws Exception {
2   Data result = null;
    NodeList list = unitNode.getChildNodes();
4   // Erster Durchlauf zum Finden des Bezeichners
    for (int i = 0; i < list.getLength(); i++) {
6     if (list.item(i).getNodeType() != Node.ELEMENT_NODE)
        continue; // wenn kein Element, dann skip
8     Node element = list.item(i);
      if (element.getNodeName().equals(id))
10      result = new Data(element.getTextContent());
    }
12  // Zweiter Durchlauf für das setzen der restlichen Eigenschaften
    for (int i = 0; i < list.getLength(); i++) {
14    if (list.item(i).getNodeType() != Node.ELEMENT_NODE)
        continue; // wenn kein Element, dann skip
16    Node element = list.item(i);
      // Position setzen
18    if (element.getNodeName().equals(position))
        result
20          .setPosition(extractContainerProperty(element,
                "Position"));
22    else if (simplePropertyList.contains(element.getNodeName()))
        if (element.getNodeName().equals("IsPlatform"))
24        result.addAbstractProperty(extractSimpleProperty(element,
              Type.TBool));
26      else
          result.addAbstractProperty(extractSimpleProperty(element,
28            Type.TString));
      else if (vectorPropertyList.contains(element.getNodeName()))
30      result.addContainerProperty(extractContainerProperty(element,
            element.getNodeName()));
32    else if (taxonomyList.contains(element.getNodeName()))
```

```
            result.addTaxonomie((extractSimpleProperty(element)));
34        else ; // skip
        }
36      return result;
    }
```

Listing 4.2: Beispielhafte Implementierung der *extractDataFromNode()*-Methode

Mapper

Bereits im vorletzten Abschnitt wurde angesprochen, dass der *Mapper* dafür Sorge trägt, dass das Datenmodell in eine dreidimensionale Darstellung überführt wird. Da nicht für jedes Problem auch dieselbe Visualisierung zielführend ist, besteht hier die Möglichkeit zur Individualisierung.

Bei der konkreten Implementierung sind zwei unterschiedliche Aspekte zu unterscheiden: Auf der einen Seite ist die Frage, wie die Daten an sich dargestellt werden, zum Beispiel in Bezug auf Form und Farbe und das sonstige Erscheinungsbild. Auf der anderen Seite muss spezifiziert werden, in welcher Art und Weise die Dimensionen der Daten dargestellt werden.

Resultat des Mappingvorgangs soll ein Szenenbaum sein. Er ist die für die Grafikengine lesbare Form des Datenmodells. Dieser Szenenbaum muss bei jeder Veränderung des Datenmodells aktualisiert werden. Als solche zählt zum Beispiel das Filtern von Daten nach bestimmten Eigenschaften oder das Erstellen einer Sicht auf das Datenmodell.

Spezialisierung Wie oben bereits beschrieben hat die Spezialisierung im *Mapper* zwei Aspekte. Auf der einen Seite kann die Gestalt eines einzelnen Datums spezifiziert werden und auf der anderen Seite die Position dieses Datums im dreidimensionalen Raum. Eine vollständige Trennung ist in der derzeitigen Version nicht gelungen. Inwiefern sich das auswirkt, wird im Folgenden gezeigt.

Position und Projektion Eine grundlegende Entscheidung bei der Visualisierung von Daten ist, wie aus den Rohdaten eines Datums die Koordinaten im dreidimensionalen Raum gewonnen werden können. Bereits in Abschnitt 4.1.1 wurde beschrieben, dass

solche Rohdaten existieren müssen. Ebenfalls ist darauf verwiesen worden, dass ungeeignete Rohdaten (weder in Gleitkomma-, noch in ganzzahliger Darstellung) an dieser Stelle durch einen geeigneten Homomorphismus umzuwandeln sind.

Der erste Schritt zur Individualisierung des *Mappers* ist die Bestimmung der Ausmaße der Projektionsfläche. Dieser Begriff meint die Ebene, die trivialisiert durch Länge und Breite aufgespannt wird. Ihre Festlegung erfolgt durch die Implementierung der *getSize()*-Methode. Hier gibt es zwei Varianten der Umsetzung. Es kann eine feste, von den Daten unabhängige Projektionsfläche gewählt werden. Eine maßstabsgetreue Abbildung entfällt somit. Vorteil dieser Art ist, dass bereits zur Implementierungszeit abschätzbar ist, wie groß die maximalen Entfernungen ausfallen. Somit können Entscheidungen, wie die Positionierung der Kamera [REF], sinnvoll getroffen werden.
Eine andere Möglichkeit ist eine von den Eingabedaten abhängige Projektionsfläche. Hier entfällt das abschätzbare Ausmaß der Darstellung zum Wohle der maßstabsgetreuen Abbildung.
Eine der wichtigsten Verwendungen dieser Methode liegt im späteren Erzeugen des Gittermusters, welches eine bessere Einschätzung der Dimensionen ermöglicht.

Der zweite Schritt zur Individualisierung ist die Festlegung der Skalierung der Position auf den einzelnen Dimensionen. Die Notwendigkeit hierfür kann sich aus ungünstig gestalteten Rohdaten ergeben. Um zum Beispiel Zeiten in Ortsangaben umzuwandeln, kann man auf die interne Darstellung eines Zeitpunktes in die Zahl der Millisekunden seit dem 01.01.1970 zurückgreifen. Hieraus ergeben sich jedoch schon für kurze Zeitspannen riesige Zahlenwerte, die zu ebensogroßen Ausmaßen der Darstellung führen würden. Mit der Implementierung der Methode *getDimensionFactors()* kann diesem Umstand Rechnung getragen werden. Ist eine Skalierung nicht notwendig, so kann die triviale Skalierung mit dem Faktor 1 erfolgen.

Der letzte Schritt zur Positionierung ist die Festlegung eines Datums auf eine genaue Koordinate in der dreidimensionalen Darstellung. Implementiert werden muss dies gleichzeitig mit der Festlegung der Gestalt eines Datensatzes. Wie dies zu erfolgen hat, ist in der allgemeinen Beschreibung zur Positionierung in Abschnitt [REF jme, Positionierung] beschrieben. Der Ort dieser Implementierung ist die im Folgenden beschriebene Methode.

Gestalt Die Gestalt eines Datums wird von der Methode *extractNodeFromData()* bestimmt. Sie hat die Aufgabe, einen Szenenknoten (siehe Abschnitt [REF]) für jeden Da-

tensatz zu erstellen. Bestandteil muss auf jeden Fall ein Grafikobjekt, wie in Abschnitt [REF, Visualisierung eines Datums] beschrieben, sein. Im letzen Abschnitt wurde bereits angesprochen, dass ebenfalls die Festlegung der Koordinate des Grafikobjekts an dieser Stelle zu erfolgen hat.
In dieser Methode kann der Benutzer neben Form, Größe, Position und sonstigen Eigenschaften, wie zum Beispiel Renderingoptionen (siehe Abschnitt [REF jme, Alphablending]), auch die Texturierung von Objekten festlegen. Es ist jedoch nicht immer zweckmäßig, für jedes Objekt eine individuelle Textur zuzuweisen. Aus diesem Grund gibt es eine weitere Methode, die es gilt zu implementieren.

Die Methode *texture()* hat zur Aufgabe die Grafikobjekte eines übergebenen Szeneknotens zu texturieren. Dies kann anhand von bestimmten Eigenschaften geschehen. Enumerationen eignen sich hierbei sehr gut. Wie in Abschnitt [REF Visualisierung eine Datums] beschrieben, sollte man sich an dieser Stelle auf eine noch vom Kurzzeitgedächtnis erfassbare Menge an Farben beschränken. Ist es jedoch nicht notwendig, so kann die Implementierung dieser Methode leer verbleiben.

Hilfsfunktionen Hilfsfunktionen der Art, wie sie aus dem *Importer* bekannt sind, gibt es im *Mapper* nicht. An dieser Stelle soll jedoch die Methode *getDataRoot()* erwähnt werden, die den Zugriff auf den vom *Mapper* erstellten Szenegraphen ermöglicht. Sie ist in zwei Ausprägungen vorhanden: Einerseits ohne Angabe von Argumenten um an die Visualisierung des Datenmodells zu gelangen, das bei der Erstellung des *Mappers* übergeben wurde. Andererseits kann auch ein Datenmodell, welches zum Beispiel durch Filtern entstanden ist, übergeben werden. Es wird automatisch zum neuen Modell des *Mappers* und daraus wird der neue Szenengraph erzeugt.

4.1.4 Zusammenspiel der Komponenten

Nachdem im letzten Abschnitt die notwendigen Möglichkeiten zur Individualisierung durch konkrete Implementierung erläutert wurden, soll im Folgenden das Zusammenspiel der Komponenten und damit die Funktionsweise des Frameworks anhand von ausgewählten Aspekten skizziert werden.

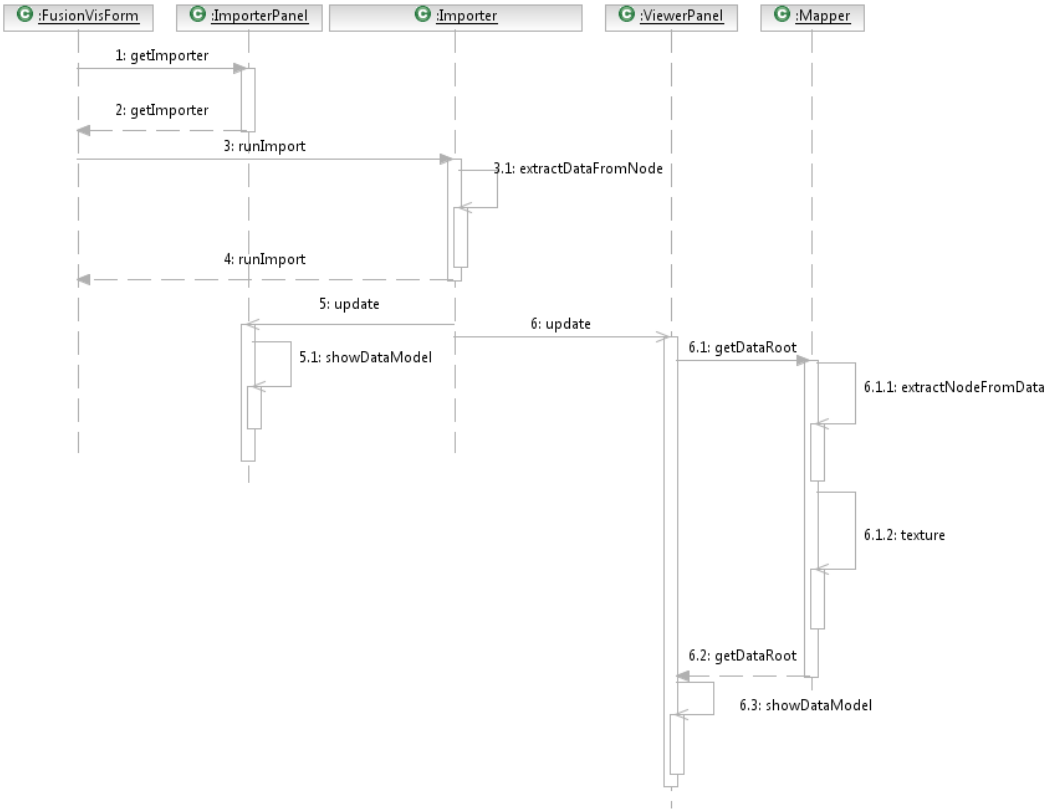

Abbildung 4.5: Ablauf des Visualisierungsprozesses

Visualisierungsprozess

Ein wesentlicher Punkt in der Interaktion der Komponenten ist der Visualisierungsprozess, der alle Komponenten des Frontends und des Backends erfasst. Hier die dazugehörige chronologische Abfolge:

1. Der Benutzer wählt eine XML-Datei aus. Die Form ruf daraufhin vom zu ihr gehörenden *ImporterPanel* den *Importer* ab.
2. Das *ImporterPanel* liefert den geforderten *Importer*.
3. Die Form stößt nun den eigentlichen Importvorgang im *Importer* an. Der *Importer* greift dabei auf die vom Benutzer implementierte *extractDataFormNode()*-Methode zurück.
4. Nach dem Anstoßen hat die Form nichts mehr mit dem restlichen Vorgang zu tun.
5. Aufgrund der Veränderung des Datenmodells durch das Laden von Daten müssen

die Sichten in den Panels aktualisiert werden. Auf der einen Seite aktualisiert das *ImporterPanel* die textuelle Darstellung der Daten.

6. Auf der anderen Seite muss das *ViewerPanel* die grafische Datensicht aktualisieren.

   a) Dazu muss das Datenmodell in einen Szenenbaum umgewandelt werden. Diese Aufgabe delegiert das Panel an den *Mapper*. Dieser erstellt mithilfe der aus dem letzten Abschnitt bekannten Methode *extractNodeFromData()* den Baum. Mit der Methode *texture()* werden die Visualisierungen der Daten anschließend eingefärbt.

   b) Den fertigen Szenenbaum gibt der *Mapper* an das *ViewerPanel* zurück.

   c) Abschließend stellt das *ViewerPanel* die Daten grafisch dar.

Der beschriebene Prozess kann in Abbildung 4.5 nachvollzogen werden.

Die angesprochene Aktualisierung der beiden unterschiedlichen Darstellungen erfolgt durch die Anwendung des Observer-Patterns (nach [GHJV02]). Dabei trägt die Form die Verantwortung, dass auch beide Panels beim Importer als Observer registriert werden. Diese Funktion ist aber bereits in der FusionVisForm implementiert, sodass der Benutzer von dieser Aufgabe unbehelligt bleibt.

ImporterPanel und ViewerPanel

Bereits im letzten Abschnitt ist deutlich geworden, dass bei Änderungen am Datenmodell zwangsläufig die Sichten aktualisiert werden müssen. Eine Situation, bei der diese Aktion notwendig ist, ist der Datenimport. Im Folgenden werde ich die beiden weiteren Aktivitäten beschreiben, bei denen sich die Panels abstimmen müssen.

Filtern des Datenmodels Über das *ImporterPanel* ist es möglich, die Sicht auf das Datenmodell zu beschränken, indem Datensätze anhand von nicht vorhandenen Eigenschaften ausgeblendet werden. Wie dies aussieht, ist in Abschnitt 4.2 beschrieben.

Das Verändern des Datenmodells durch Ausgrenzen bestimmter Datensätze macht die Aktualisierung der Sicht im *ViewerPanel* notwendig. Dies geschieht auf die gleiche Art und Weise wie ein Neuimport von Daten, denn prinzipiell ist das Filtern von Daten nur

die Gegenoperation zum Import und somit nur eine Veränderung am Datenmodell. Aus diesem Grund wird auch an dieser Stelle das Observerpattern genutzt.

Problematisch ist hierbei, dass die beiden Panels bereits eine Oberklasse haben und somit nicht von der in Java bereits zur Verfügung gestellten Observerklasse erben können. Ein Ausweg aus diesem Problem wird in Abschnitt 4.3.1 dargestellt.

Selektion von Datensätzen Eine weitere Aktivität, in der die beiden Panels kooperieren müssen, ist die Selektion eines Datensatzes. Ein Datum kann entweder im *Importer-Panel*, in der textuellen, oder im *ViewerPanel*, in der grafischen Darstellung, ausgewählt werden.

Ziel soll es sein, dass ein Datum, das in der einen Sicht ausgewählt wird, auch in der andern Sicht als selektiert erscheint. Die Auswahl an sich ist im *ImporterPanel* aufgrund der Bordmittel von Swing [REF Technologie, Java, SWING] trivial. Im *ViewerPanel* wird dies um einiges schwieriger, weil es sich dabei um einen sogenannten Mousepick handelt. Problematisch ist dabei, dass auf der zweidimensionalen Fläche des Bildschirms ein Objekt im dreidimensionalen Raum ausgewählt werden soll. Für die Lösung des Problems sei auf Abschnitt 4.3.4 verwiesen.

Nachdem ein Datensatz ausgewählt wurde, wird auch an dieser Stelle wieder mithilfe des Observer-Patterns das jeweils andere Panel über das Ereignis informiert, sucht in seiner Darstellung den selektierten Datensatz und hebt ihn hervor.

Die Kopplung der beiden Datenbestände erfolgt über den Bezeichner des Datums. Bereits in Abschnitt 4.1.1 wurde gefordert, dass es sich hierbei um eine eindeutige Eigenschaft handeln muss. Wäre dies nicht der Fall, ist das geforderte Ziel der abgestimmten Auswahl schlicht nicht zu erreichen.

ViewerPanel und Viewer

Aus der Abbildung 4.4 wird deutlich, dass der Viewer aus dem allgemeinen Rahmen der Struktur heraus fällt, da er als innere Klasse des ViewerPanels umgesetzt ist. Dies ist ein aus entwurfsästhetischen Gesichtspunkten unschöner Sachverhalt.
Betrachtet man jedoch die enge Zusammenarbeit zwischen dem Panel und dem Viewer, wird klar, warum die pragmatische zugunsten der ästhetischen Lösung gewählt wurde.

Zum einen sollte das Panel vor allem in Hinblick auf Erweiterbarkeit, wie zum Beispiel einstellbare Kameraperspektiven und eine Schnittstelle zum Viewer bieten. Wäre dieser als separate Klasse umgesetzt, hätten viele Methoden und Instanzvariablen publik gemacht werden müssen. Eine geeignete Kapselung wird so unmöglich. Aus diesem Grund bietet es sich an, den Viewer zwecks der Abschirmung in das Panel einzubinden.

Ein anderer Aspekt, bei dem genau derselbe Zwiespalt auftritt, ist das Auswählen von Daten im ViewerPanel. Da der Aufbau von Komponenten in Swing hierarchisch organisiert ist, klickt der Benutzer beim Selektieren eines Objekts in der dreidimensionalen Darstellung nicht auf die vom Viewer bereitgestellte Zeichenfläche, sondern in die Form, die wiederum nativ das Ereignis an das zuständige Panel weitergibt.

An dieser Stelle bricht jedoch die Kette, denn der Viewer ist nicht in die Swing- Klassenstruktur eingebettet. Aus diesem Grunde kümmert sich das ViewerPanel um die für das Auswählen notwendigen Schritte. Zu diesem Zweck benötigt es ähnlich wie oben angesprochen einen weitreichenden Zugriff auf den Viewer. Ohne ihn komplett öffentlich zu gestalten führt auch hier der Weg an einer inneren Klasse nicht vorbei.

Aus Sicht der Unterscheidung Framework-Komponenten in Frontend und Backend ist die Zuordnung des Viewers in das Panel vertretbar, denn er ist mit seiner Zeichenfläche im Gegensatz zum Importer eine sichtbare Komponente.

4.2 Visualisierung einer militärischen Lage

Bereits in Abschnitt 1.1 wurde die Lage des S2-Offiziers angesprochen, der von den Panzern seines Bataillons Positions- und Feindmeldungen bekommt. Dies geschieht aus Gründen der Datenübertragung in textueller, beschränkt menschenlesbarer Form.

4.2.1 Problemdarstellung

Die Aufgabe des S2-Offiziers ist es, aus diesen Daten ein Lagebild zu erstellen, also die Daten zu visualisieren. Problematisch sind hierbei vor allem die Feindmeldungen. Folgende Lage veranschaulicht das Problem:

In der Norddeutschen Tiefebene treffen zwei eigene Panzer auf zwei gegnerische. Die eigenen Einheiten bewegen sich nicht, sie stehen in teilgedeckter Stellung. Die feindlichen

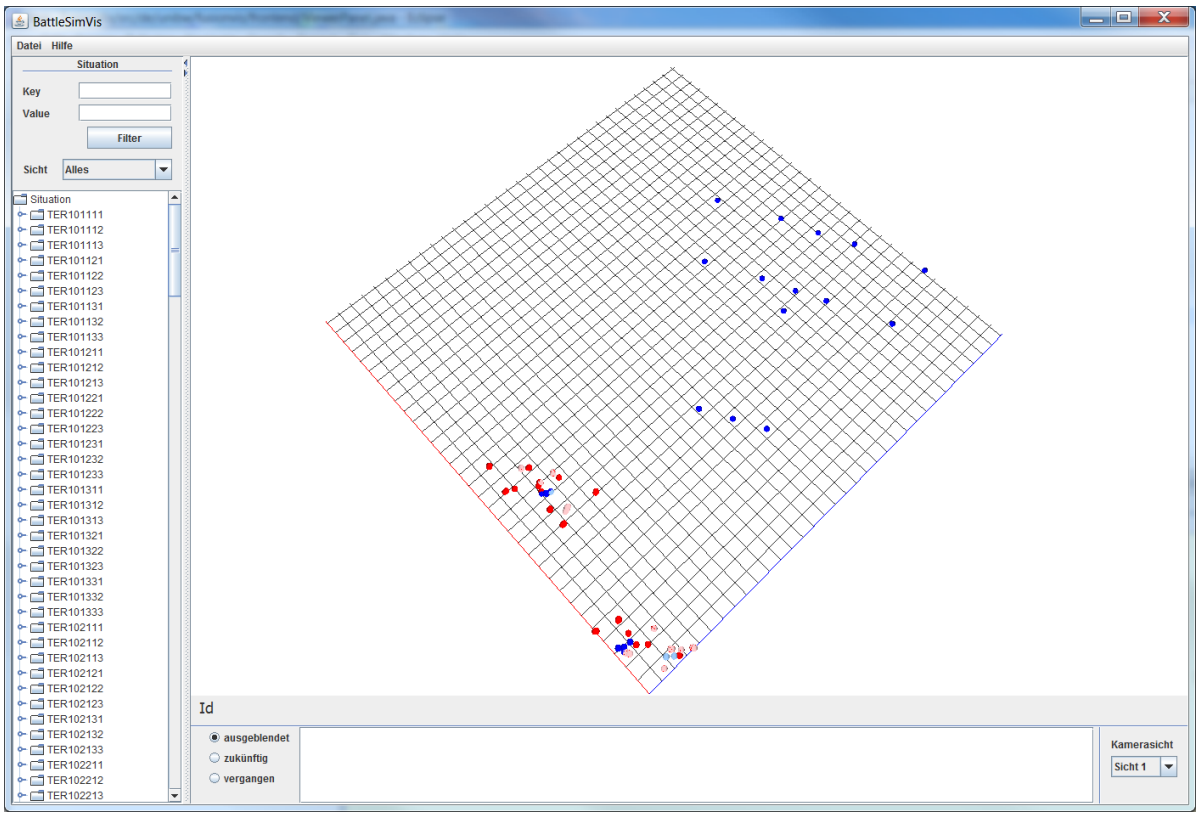

Abbildung 4.6: Darstellung einer Lage mit zwei aufeinandertreffenden Bataillonen, insgesamt 251 Datensätze

Einheiten bewegen sich mit hoher Geschwindigkeit, eine in Richtung auf die beiden blauen[1] Panzer, die andere in Querfahrt von Rechts nach Links aus Sicht der beobachtenden Einheiten.

Im Abstand von 60 Sekunden geben die beiden eigenen Panzer drei Mal nacheinander Meldungen über die gesichteten Feinde ab. Daraus resultieren zwölf Feindmeldungen.

Für den Offizier, der nun die Daten erhält, stellen sich potentiell zwölf Feindeinheiten dar. Von der Anzahl ergäbe das die Stärke einer Kompanie. Dies entspricht nicht der Wirklichkeit. Das Problem, das es nun zu lösen gilt, ist durch geeignete Visualisierung aus der vermeintlichen Kompanie wieder die zwei eigentlichen Feindpanzer erkennen zu können.

[1] Im Kontext von militärischen Lagen spricht man von blauen und roten Einheiten. Rot steht hierbei für feindliche, blau für eigene oder verbündete Einheiten.

4.2.2 Lösungsansatz

Die zahlenmäßige Explosion der potentiellen Feindmeldungen kann durch zwei Ansätze reduziert werden:

Ortsgleichheit

Als erstes gilt es, dass alle Meldungen, die zur selben Zeit gemacht wurden, auf die Lage des gemeldeten Objekts überprüft werden. Wurde an einem Ort (mit einer gewissen Toleranz) gleichzeitig mehrere Einheiten gemeldet, so kann das daher kommen, dass diese eben von mehreren eigenen Panzern gleichzeitig gesehen wurden.

Mit diesem Ansatz können die zwölf Meldungen auf nur noch vermeintlich sechs Feindeinheiten reduziert werden.

Reichweitenberechnung

Der nächste Schritt ist die Analyse der Meldungen, die zu unterschiedlichen Zeiten abgegeben wurden. Unter Annahme der maximalen Geschwindigkeit kann man nämlich feststellen, ob zwei Meldungen definitiv von unterschiedlichen Feindeinheiten ausgelöst wurden, oder ob die Möglichkeit besteht, dass es sich um ein und dasselbe Objekt handelt.

Gegeben seien dafür zwei Meldungen. Aus ihnen lassen sich die Zeitdifferenz und die Entfernung der beiden voneinander bestimmen. Ist ihre Entfernung größer, als die Distanz, die in der errechneten Zeit unter Annahme der Höchstgeschwindigkeit hätte zurückgelegt werden können, so handelt es sich definitiv um unterschiedliche Einheiten.

Auf diese Art und Weise kann die Zahl der potentiellen Einheiten weiter der Zahl der echten Einheiten angenähert werden und ein aussagekräftigeres Lagebild entsteht.

4.2.3 Umsetzung mithilfe des Visualisierungsframeworks

Dass die oben dargestellten Lösungsansätze mithilfe von Berechnungen umgesetzt werden können, steht außer Frage. Mit dem im Folgenden vorgestellten Prototypen soll aber

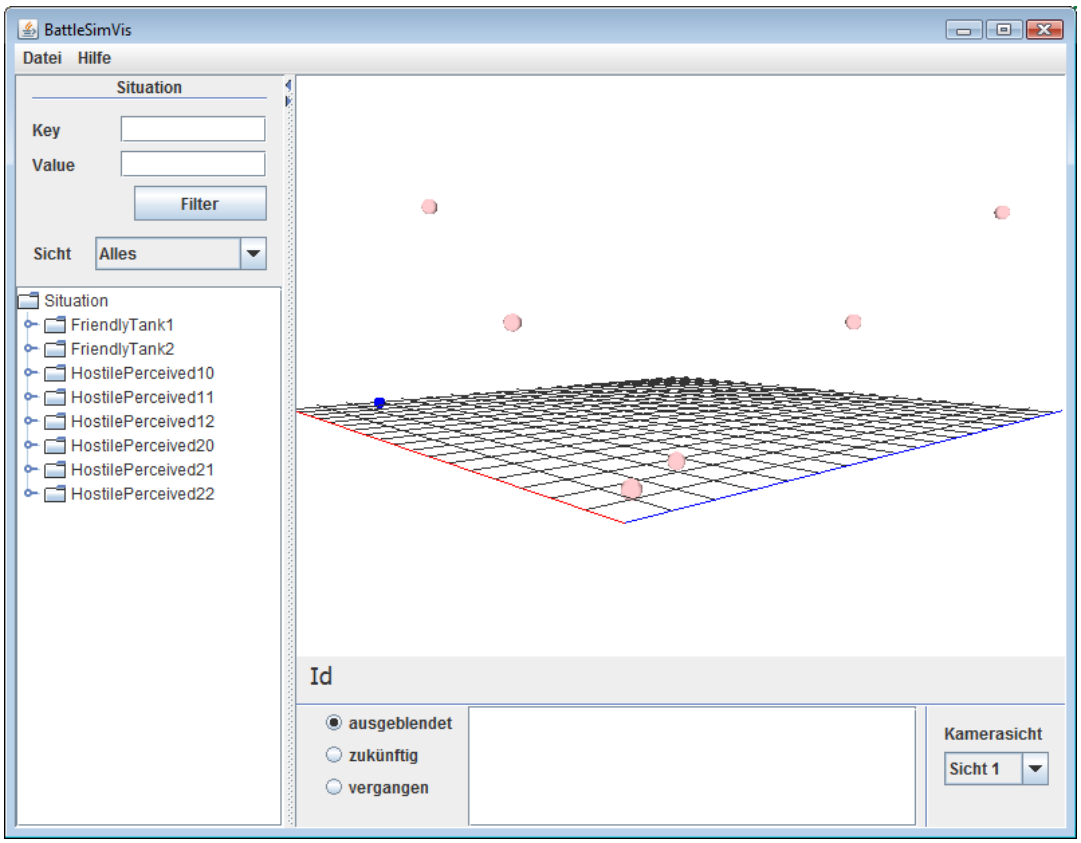

Abbildung 4.7: BattleSimVis als prototypische Implementierung einer militärischen Lagedarstellung

ein anderer Ansatz verfolgt werden: Der S2-Offizier soll mithilfe des umgesetzten Programms allein durch die Betrachtung der visualisierten Darstellung aus den redundanten Daten ein der Realität nahe kommendes Lagebild erstellen können.

Aufbau der Benutzerschnittstelle

Die Benutzeroberfläche ist in zwei Teile gegliedert. Auf der einen Seite gibt es die textuelle Sicht auf die Lage und alle Möglichkeiten, diese zu manipulieren. Auf der anderen Seite sind die grafische Lagendarstellung in dreidimensionaler Form und die Mittel, diese zu beeinflussen. Welche das im Detail sind, wird im Folgenden beschrieben. Grundlage dieser Aufteilung ist die Struktur, die durch das Visualisierungsframework vorgegeben wird (siehe Abschnitt 4.1.2).

Auswahl der Dimensionen

Um die folgenden Überlegungen und die Grundidee hinter der Visualisierung der Lage im Prototypen zu verstehen, ist es notwendig, zu erklären, welche Dimension im dreidimensionalen Raum mit welcher Größe der Daten übereinstimmt.

Die Ebene, aufgespannt durch Länge und Breite, spiegelt die Position eines Objekts im Raum wieder. Dabei ist die Höhe vernachlässigt worden. Da die Eingabedaten in geographischer Länge und Breite angegeben sind, kommt es hier zu einem, durch die Projektion einer Kugelkappe in die Ebene entstandenen Fehler bezüglich der maßstabsgetreuen Darstellung von Entfernungen. Wie stark sich dieser auswirkt wird in Abschnitt 4.3.3 erläutert.

Die dritte Dimension, die man so durch die Vernachlässigung der Höhe gewinnt, kann durch die Zeit eingenommen werden. Meldungen, die am ältesten sind, werden direkt auf der durch Gitternetzlinien angezeigten Grundebene dargestellt. Die Anzeige jüngerer Meldungen erfolgt kontinuierlich inpositive y-Richtung (*in die Höhe*).

Der Grund für diese Auswahl ist die später beschriebene Möglichkeit, die Reichweite einer Einheit in Abhängigkeit von ihrer Höchstgeschwindigkeit darzustellen.

Umgesetzte Fähigkeiten

Plattformunabhängigkeit *BattleSimVis* ist plattformunabhängig. Es wurde entwickelt um auf allen gängigen Betribssystemen zu funktionieren. Getestet ist es mit MS Windows (XP, Vista, 7), MacOS X, Linux (Ubuntu 9.10). Einzige Voraussetzung neben einem aktuellen Java Runtime Environment ist eine 3D-fähige Grafikkarte mit zugehörigen Treibern.

Import der Daten: *BattleSimVis* ist durch das verwendete Visualisierungsframework in der Lage, Daten im XML-Format zu lesen. Für den Testgebrauch wurde eine XML-Datenstruktur verwendet, die aus einem Simulationsproxy [BELEG] stammt. Die importierten Daten werden in eine textuelle und eine grafische Ansicht umgewandelt.

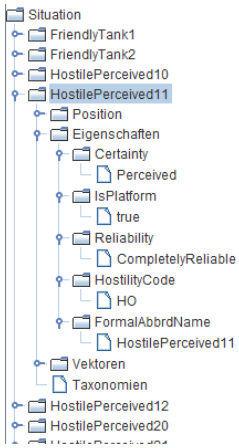

Abbildung 4.8: Baumstruktur der textuellen Darstellung

Baumartige Datenanzeige: Die textuelle Anzeige der Daten erfolgt in einer Baumstruktur. Auf erster Ebene befinden sich die Datensätze, dargestellt durch ihren Bezeichner. Darunter sind für jeden Datensatz die eingelesenen Eigenschaften dargestellt.

Dreidimensionale Datenanzeige: Im dafür vorgesehenen Bereich des Prototypen werden die gelesenen Datensätze im dreidimensionalen Raum dargestellt.

Freie Navigierbarkeit: Durch einen Klick in die grafische Visualisierung ist es dem Benutzer möglich, frei durch den dreidimensionalen Raum zu navigieren. Durch Druck der mittleren Maustaste und Bewegung der Maus kann die Ansicht frei gedreht werden. Die Position des Betrachters wird, ähnlich zu der Steuerung von Computerspielen aus der Ego-Perspektive, mit den Tasten [W], [A], [S] und [D] gesteuert.

Maßstabsgetreue Abbildung: Bis auf die oben gemachte Einschränkung bezüglich des Fehler der Projektion einer Kugelkappe in die Ebene, werden die Daten maßstabsgetreu dargestellt. Die Berechnung der Entfernung erfolgt, aufgrund der Tatsache, dass die eigentlichen Koordinaten die Position der Einheit auf einer Kugel beschreiben, durch die in Abschnitt 4.3.2 dargestellte Haversine-Formel.
Die Gitternetzlinien sind auf die importierten Daten angepasst und werden nur in einem notwendigen Bereich auch angezeigt.

Farbige Darstellung der Eigenschaften: Um eine militärische Lage zu überblicken, ist die Kenntnis über die Zuordnung der Einheiten zu Freund und Feind unabdingbar. Aus diesem Grund stellt *BattleSimVis* feindliche Einheiten in rot und freundliche in blau dar. Weiterhin wird über die Helligkeit der Farbe unterschieden, ob es sich um eine echte Einheit (ein meldender eigener Panzer) oder um eine Meldung (einen feindlichen Panzer) handelt. Eine Meldung ist heller, eine echte Einheit wird in einem satten Farbton dargestellt.

Voreingestellte Kameraperspektiven: Zusätzlich zu der freien Navigierbarkeit in der dreidimensionalen Darstellung gibt es die Möglichkeit, zwischen drei vordefinierten Perspektiven auszuwählen. Zwei zeigen gegenüberliegende isometrische Ansichten, die dritte stellt die Lage aus einer Draufsicht dar.

Datenfilter: Um die Meldungen auf ein gewünschtes Maß zu reduzieren, gibt es die Möglichkeit, die importierten Daten nach bestimmten Eigenschaften auszuwählen. Dafür verfügt *BattleSimVis* über einen Filter, der anhand von Schlüssel-Wert-Paaren Datensätze auswählt. Die Schlüssel stehen für bestimmte Eigenschaften der Meldungen (Freund-Feind-Kennung, Verlässlichkeit etc.).

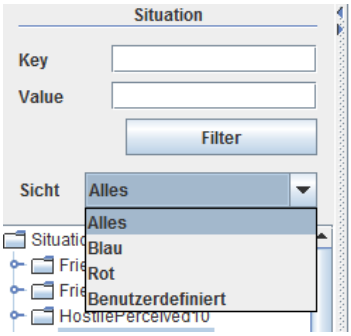

Abbildung 4.9: Filter mit voreingestellten Sichten auf die Lage

Vordefinierte Sichten auf die Lage: Um eine Lage (unabhängig von der oben geschilderten Situation des S2-Offiziers) aus der Sicht der roten Einheiten, der blauen Einheiten oder omniscient anzusehen, gibt es vordefinierte Filter. Sie zeigen bei der Wahl der blauen Sicht zum Beispiel alle echten blauen Einheiten, aber nur alle gemeldeten roten dar. Im Fall der Feindsicht ist dies genau andersherum. Die omnisciente Sicht zeigt alle Datensätze.

Selektion in der grafischen Darstellung: Um die textuelle und die dreidimensionale Sicht optimal miteinander verbinden zu können, ist es erforderlich, dass eine Auswahl einer Einheit (zum Beispiel zum Inspizieren ihrer Eigenschaften) in beiden Darstellungen vorgenommen werden kann. In der Baumstruktur ist dies trivial durch einen Klick auf den Bezeichner und das damit verbundene Aufklappen des Baums möglich.
Um auch in der grafischen Ansicht die Eigenschaften zu einer Einheit zu erfahren, klickt man sie dort einfach an. Durch einen durchscheinenden grünen Kasten wird sie hervorgehoben und in der Baumstruktur wird der dazugehörige Pfad expandiert und ausgewählt. Dieses Fähigkeit wird als Mousepicking bezeichnet und wird in Abschnitt 4.3.4 näher erläutert.

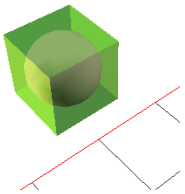

Abbildung 4.10: Hervorhebung in der grafischen Ansicht

Bewegungskegel: Eine Kernfähigkeit von *BattleSimVis* ist das Einblenden von Bewegungskegeln. Die oben beschriebene Auswahl, welche Eigenschaft einer Meldung auf welche Dimension abgebildet wird, vor allem die Wahl, die Zeit in die Höhe zu zeichnen, ermöglicht die Visualisierung der Reichweite einer Einheit.
Wird eine Meldung in der grafischen Darstellung ausgewählt, kann ein Kegel ein- und ausgeblendet werden, der anzeigt, wo sich eine Einheit in der Zukunft aufhalten kann (nach oben geöffneter Kegel) oder in der Vergangenheit aufgehalten haben könnte. Alles was innerhalb dieses Kegels liegt, ist in Reichweite der Einheit unter Berücksichtigung ihrer Maximalgeschwindigkeit. Alles was außerhalb ist, kann nicht in der jeweiligen Zeit erreicht worden sein.

Inhalt von Bewegungskegeln: Auch wenn oben beschrieben wurde, dass der S2–Offizier mithilfe von *BattleSimVis* allein durch das Ansehen einer Lage Redundanzen ausschalten kann, unterstützt das Tool den Anwender bei der Auswertung von Bewegungskegeln, indem es eine Liste aller Einheiten anzeigt, die sich darin befinden.

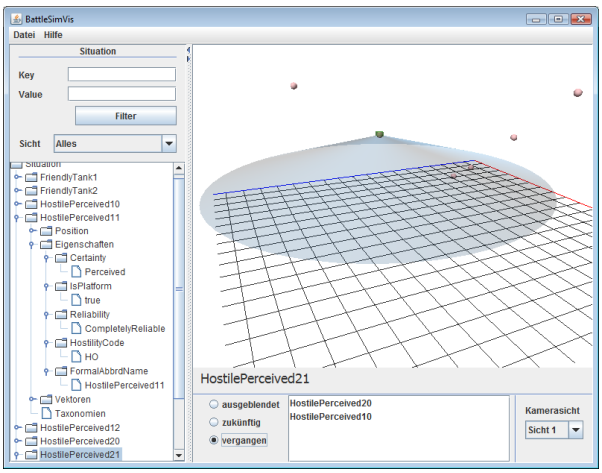

Abbildung 4.11: Bewegungskegel in die Vergangenheit

Nutzung des Tools zur Lösung des gestellten Problems

Mit den oben dargestellten Fähigkeiten von *BattleSimVis* ist es dem S2-Offizier nun möglich, die beiden diesem Abschnitt aufgezeigten Lösungsansätze zu nutzen, um ein geeignetes Bild von der Lage zu bekommen.

Das Problem der mehrfachen Meldung von Feindeinheiten durch eigene Kräfte, die gleichzeitig zum Beispiel einen roten Panzer entdecken, wird vom Programm dadurch behoben, dass die Einheiten auch am selben Ort dargestellt werden. Idealerweise sieht der S2-Offizier somit nur einen Panzer, wo auch nur ein Panzer in der Realität steht. Weichen die Daten geringfügig ab, fällt dies bei Berechnungen auf, jedoch erkennt man Unterschiede in der Position beim Ansehen erst bei so signifikanten Entfernungen, dass es sich um unterschiedliche Einheiten handeln muss.

Um eine Reichweitenanalyse durchzuführen, also um zwei zu unterschiedlichen Zeitpunkten gemeldete Objekte eindeutig als auch in der Realität nicht identisch zu identifizieren, wird dem S2-Offizier die Möglichkeit an die Hand gegeben, zu jeder Einheit einen in die Zukunft oder einen in die Vergangenheit ragenden Trichter anzeigen zu können. Damit lässt sich die theoretische Reichweite eines Objekts anhand der Maximalgeschwindigkeit visualisieren. Alles, was nicht in diesem Trichter ist, muss auch eine andere Einheit sein.

Somit ist gezeigt, dass die prototypische Implementierung durch die gewählte Visualisierung die Reduktion redundanter Daten ermöglicht und den S2-Offizier bei der Fusion multisensorischer Daten adäquat unterstützt.

4.3 Kernaspekte der Implementierung

Bei der Umsetzung von Framework und einer prototypischen Implementierung fanden sich einige Implantierungsdetails, die es wert sind, auf sie genauer einzugehen. Dies kann drei Gründe haben: Entweder sie sind für das Verständnis der Umsetzung relevant (zum Beispiel Abschnitt 4.3.4), sie beinhalten elegante mathematische Ansätze (siehe Abschnitt 4.3.5) oder erläutern andere mögliche Ansätze, die aus genannten Gründen nicht verfolgt wurden.

4.3.1 Observer-Pattern bei abgeleiteten Klassen

Bei der Auswahl eines Datums in der textuellen Ansicht soll die resultierende Hervorhebung auch in der grafischen Darstellung erfolgen und umgekehrt. Zur Umsetzung dieser Aufgabe empfiehlt sich das Observer-Pattern.

Dabei gibt es auf der einen Seite eine observierbare Klasse. Dies ist der Ausgangspunkt, an dem die Veränderung eines Zustands auftritt. Sie verwaltet eine Liste von Klasseninstanzen, die an dieser Veränderung interessiert sind. Geschieht ein Zustandsübergang, werden diese von der observierten Klasse benachrichtigt und können angemessen reagieren.

Dieses Entwurfsmuster (nach [GHJV02]) ist in Java bereits in die mitgelieferte Klassenbibliothek integriert. Dabei ist der observierte Anteil als Klasse *Observable* fertig umgesetzt. Ein Datenmodell zum Beispiel, das beobachtet werden und in unterschiedlichen Ansichten dargestellt werden soll, wird von der *Observable*-Klasse abgeleitet. Alle notwendigen Methoden werden somit zur Verfügung gestellt.

Der beobachtende Anteil ist in Java als Interface umgesetzt. Es existiert somit nur ein Gerüst von Methoden. In diesem Fall ist es nur eine einzige, die *update()*-Methode. In ihr muss das Verhalten implementiert werden, das ausgelöst wird, wenn sich an der beobachteten Instanz etwas ändert.

Dieser von Java bereitgestellte Ansatz kommt an seine Grenzen, wenn die beobachtete Klasse bereits von einer anderen erbt. Denn dann kann sie nicht auch noch von der *Observable* abgeleitet werden. Java verbietet schließlich die Mehrfachvererbung von Klassen.

Genau dieses Problem tritt auch bei den beiden Panels ein, die bei der Auswahl von Datensätzen wechselseitig sowohl Beobachter, als auch Beobachteter sind. Sie sind beide von der Swing-Klasse *JPanel* abgeleitet und dürfen nicht von *Observable* erben.

Zu diesem Problem gibt es zwei unterschiedliche Lösungsansätze. Auf der einen Seite kann man ein *Observable*-Interface erstellen, muss dann aber die Liste der Beobachter und die Methoden zum Benachrichtigen selbst implementieren. Dies ist ein nicht unerheblicher Aufwand und zudem unter Umständen fehlerträchtiger Aufwand.

Der andere Ansatz ist eine innere Klasse in den zu Beobachtenden Klassen zu erstellen, die dadurch von *Observable* erben kann und als innere Klasse den vollen Zugriff auf die Instanzvariablen ihrer Umgebung hat. Auf der anderen Seite kann die äußere Klasse die Methode zum benachrichtigen der Beobachter zum geeigneten Zeitpunkt aufrufen.

So kann mit nur geringen Anpassungen die vorimplementierte Lösung übernommen und das Problem gelöst werden.

4.3.2 Entfernungsberechnung auf der Erde

Der in Abschnitt 4.2 vorgestellte Prototyp zur Visualisierung eines Schlachtfeldes berechnet den Projektionsbereich, indem die minimale und maximale geografische Länge und Breite der Eingabedaten bestimmt wird. Diese vier Daten beschreiben ein Rechteck, dessen Kantenlängen berechnet und in die dreidimensionale Darstellung übertragen werden.

An dieser Stelle ergibt sich ein Problem aus der Gestalt der Erde in Verbindung mit der Art der Positionsdaten. Weil die Lage der Objekte in Longitude (geografische Länge) und Latitude (geografische Breite) beschrieben wird, handelt es sich um Koordinaten auf einer Kugel. Daraus ergeben sich Besonderheiten für die Längenberechnung.

Weiterhin ist die Erde in Wirklichkeit aber keine Kugel, sondern ein Rotationsellipsoid, der an den Polen einen geringeren Radius hat, als am Äquator. Diese Tatsache sei aber für die folgenden Betrachtungen zu vernachlässigen. Grund dafür ist die bereits bei der Visualisierung gemachte Vereinfachung, die Höhe der Eingabedaten (siehe Abschnitt 4.3.3) zu vernachlässigen.

Zur Entfernungsberechnung sollen drei Ansätze aufgezeigt und auf ihre Tauglichkeit untersucht werden. Grundlage hierfür ist eine Frage aus der U. S. Census Bureau Geographic Information Systems FAQ ([Bur09]). Im Hinterkopf sollten dabei folgende Ausmaße

bleiben, die vermitteln, in welchen Größenordnungen man sich bewegt. Der größte Truppenübungsplatz Europas in Bergen ([Wik09b]) hat eine Ausdehnung von 25km x 18km. Er liegt etwa auf 52° nördlicher Breite. Ein Kampfpanzer vom Typ Leopard 2A5 hat eine Länge von 7,72m ([Wik09a]).

Pythagoras

Der erste Ansatz ist die naive Anwendung des Satzes von Pythagoras. Ausgangspunkt dafür ist die Annahme, sich nicht auf der Oberfläche einer Kugel zu bewegen, sondern auf einer Ebene. Dies hat Ungenauigkeiten zur Folge, die in 4.3.3 beschrieben sind. Der Abstand d zweier Punkte berechnet sich somit wie flogt:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.3.1}$$

Der Abstand hat dieselbe Einheit, wie die in kartesischen Koordinaten angegebenen Punkte *P1(x1, y1)* und *P2(x2, y2)*.

Abhängig von der geografischen Breite, an der die Berechnung durchgeführt wird, resultieren die in Tabelle 4.2 dargelegten Fehler bei einer tatsächlichen Länge von 20km.

absoluter Fehler	*geografische Breite*
< 30m	< 70°
< 20m	< 50°
< 9m	< 30°

Tabelle 4.2: Absolute Fehler bei der Entfernungsberechnung nach Pythagoras, Abstand der Punkte ist 20km

Da die für die Veranschaulichung gewählte Länge durchaus in der Realität, zum Beispiel auf einem Truppenübungsplatz auftreten kann, sind die entstehenden Fehler nicht hinnehmbar.

Seitenkosinussatz

Um die Entfernung zweier Punkte auf einer Kugel mathematisch korrekt zu berechnen, behilft man sich der sphärischen Geometrie. Für das in Abbildung 4.12 dargestellte

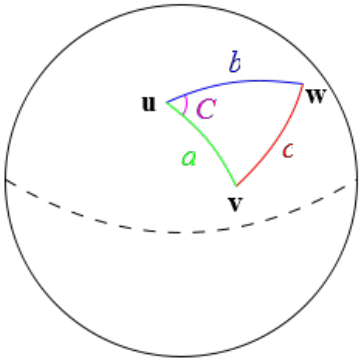

Abbildung 4.12: Kugeldreieck mit Beschriftung

Kugeldreieck gilt der Seitenkosinussatz:

$$\cos c = \cos a \cos b + \sin a \sin b \cos C \tag{4.3.2}$$

Dabei soll die Entfernung c von Punkt v nach Punkt w berechnet werden.
Nimmt man für den Punkt u den Nordpol an, und arbeitet auf einer Einheitskugel, dann folgt:

$$a = 90° - lat_1 \tag{4.3.3a}$$
$$b = 90° - lat_2 \tag{4.3.3b}$$

Bekannt ist folgender Zusammenhang:

$$\sin 90° - x = \cos x \tag{4.3.4a}$$
$$\cos 90° - x = \sin x \tag{4.3.4b}$$

Daraus ergibt sich nach Umformung eine Formel zur Berechnung der Länge. Das Ergebnis hängt von den Einheiten der Eingabedaten ab. Werden die Koordinaten im Gradmaß angegeben, so ist zusätzlich zum Erdradius mit dem Faktor $\frac{\pi}{180°}$ zu multiplizieren. Werden die Daten im Bogenmaß eingegeben, entfällt der letzte Faktor.

Der so berechnete Abstand ist unter Berücksichtigung der gemachten Vereinfachung, auf einer Kugel zu rechnen, mathematisch korrekt. Ideal ist diese Lösung jedoch nicht, da die

Formel für sehr kleine Entfernungen schlecht konditioniert ist. Kritisch ist hier vor allem der Kosinus der Längengraddifferenz. Abhängig von der Gleitkommapräzision (weniger als sieben signifikante Stellen) können bereits Längen kleiner einer Bogenminute nicht mehr unterschieden werden. Der Grund hierfür ist in Tabelle 4.3 illustriert.

x	$\cos x$
1 Bogenminute	0,99999995769202532795126248717334
30 Bogensekunden	0,99999998942300627605141750356948
1 Bogensekunde	0,99999999998824778473047407621793

Tabelle 4.3: Schlechte Konditionierung des Kosinus

Haversine Formel

Die gleiche Berechnung wie oben kann durch geeignetes Umstellen der Gleichung auch ohne das geschilderte Problem durchgeführt werden. Grundlage bilden die Additionstheoreme und die heute weniger gebräuchliche trigonometrische Funktion des Semiversus (im englischen Haversine, daher der Name der Formel).

$$d = 2R \arcsin \sqrt{\sin^2 \frac{lat_2 - lat_1}{2} + \cos lat_1 \cos lat_2 \sin^2 \frac{\Delta lon}{2}} \tag{4.3.5}$$

Auch diese Formel ist für ein spezielles Problem schlecht konditioniert. Problematisch sind Entfernungen zwischen antipodalen Punkten, also Orten, die sich auf dem Erdball gegenüberliegen. Da dieser Fall in Kontext des erstellten Prototyps keine Rolle spielt, ist dieser Fall auch zu vernachlässigen.
Aus diesem Grund wurde diese Formel zur Entfernungsberechnung verwendet.

4.3.3 Informationsverlust durch Projektion einer Kugelkappe auf eine Ebene

Um im vorgestellten Prototyp eine Dimension für die Zeit verwenden zu können, war es notwendig die Höhe einer Einheit außer Acht zu lassen. Dadurch ergibt sich eine Ungenauigkeit, denn die Eingabedaten befinden sich eigentlich auf einer sogenannten Kugelkappe. So bezeichnet man einen Ausschnitt einer Kugeloberfläche. Durch das Fortlassen der Zeit wird dieser in die Ebene projiziert.

Um diesen Sachverhalt einfacher durchdenken zu können, soll er in den zweidimensionalen Raum übertragen werden. Folglich geht es dann nicht mehr um eine Kugelkappe, die in eine Ebene abgebildet wird, sondern um einen Kreisbogen, der auf eine Gerade projiziert wird.

Durchdenkt man sich diesen Vorgang, ist es klar, dass die resultierende Gerade kürzer sein muss, als der Kreisbogen. Um diesen Fehler zu veranschaulichen soll noch einmal die Größenordnung des Truppenübungsplatzes Bergen herangezogen werden. Mit Ausmaßen von 25km x 18km ist die theoretisch längste Strecke die Diagonale des aufgespannten Rechtecks. Diese ist nach Pythagoras ca. 30km lang.

Diese 30km als Länge eines Kreisbogens auf einem Kreis mit einem Radius von 6371km sind der Anhalt für die folgende Rechnung. Verbindet man Anfang und Ende des Kreisbogens mit einer Gerade, so ist diese Strecke die gesuchte Projektion.

Zwischen dem Mittelpunkt des Kreises und den Endpunkten des Kreisbogens wird ein gleichschenkliges Dreieck aufgespannt. Das Lot auf der Strecke, dessen Länge gesucht ist, Teilt das Dreieck in zwei gleichgroße rechtwinklige Teildreiecke. Grund hierfür ist, dass das Lot auf der Grundseite eines gleichschenkligen Dreiecks zugleich auch Winkelhalbierende ist.

Folglich ist der Winkel α, der vom Lot und dem Radius, der Mittelpunkt und einen Endpunkt des Kreisbogens verbindet wie folgt zu berechnen:

$$\frac{2\pi \cdot 6371km}{2\pi} = \frac{30km}{2\alpha} \tag{4.3.6a}$$

$$\alpha = \frac{15}{6371} \tag{4.3.6b}$$

Mit dem berechneten Winkel lässt sich auf die Länge der gesuchten Strecke schließen. Sie ist das Doppelte der Gegenkathete zum Winkel α.

$$\sin\alpha = \frac{a}{6371km} \tag{4.3.7a}$$

$$l = 2\sin\alpha \cdot 6371km \tag{4.3.7b}$$

Die Länge der Strecke ist somit etwa 29,999972km. Der absolute Fehler beträgt 2,72cm. Relativ sind das 0,001 ‰. Das Vernachlässigen der Höhe fällt also viel schwerer ins Gewicht, als die beschriebene Vereinfachung.

4.3.4 Mousepicking

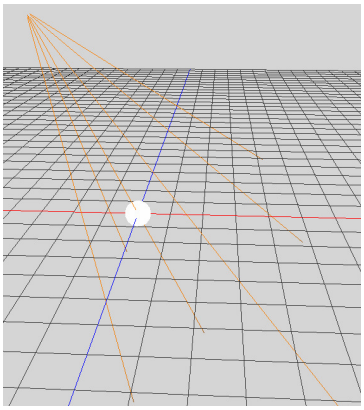

Abbildung 4.13: Darstellung der von der Maus ausgehenden Strahlen aus [Sch08]

Unter dem Wort Mousepicking wird nach [Sch08] die Auswahl von Objekten in einer dreidimensionalen Darstellung verstanden. Dies ist in der Umsetzung des Visualisierungsframeworks notwendig, wenn ein Datum in der grafischen Ansicht selektiert werden soll, um zum Beispiel seine Eigenschaften in der textuellen Ansicht zu inspizieren.

Bei dieser, auf den ersten Blick sicher trivial erscheinenden Tätigkeit gibt es ein Problem, dass der Leser vielleicht aus der Bäckerei kennt. Zeigt man, vor der Theke stehend, auf ein Gebäckstück hinter der Glasscheibe, welche die Backwaren vor den Kunden schützt, so ist es für den/die Verkäufer/in nicht einfach zu erkennen, auf was gezeigt wird.

Ein ähnlicher Sachverhalt ergibt sich durch das Zeigen mit der Maus auf ein Objekt im dreidimensionalen Raum, denn der Mauszeiger bewegt sich nur auf einer Ebene, einer Art Fenster zum Raum der Darstellung. Um zu berechnen, auf was gezeigt wird, braucht es einen Strahl ausgehend von der Spitze des Mauszeigers. Dasjenige Objekt, das in der kürzesten Entfernung vom Zeiger getroffen wird, soll ausgewählt werden.

Aus der analytischen Geometrie ist aber bekannt [CITE], dass eine Gerade, spezieller aber auch ein Strahl, durch mindestens zwei Punkte zu spezifizieren ist. An dieser Stelle hilft die gewählte Grafikengine, indem sie ihrem Benutzer zu einer gewählten Bildschirmkoordinate den Startpunkt des Strahls, der von ihr ausgeht, im dreidimensionalen Raum liefert. Zusätzlich ist sie in der Lage, senkrecht zur Bildschirmebene einen Zielpunkt am Ende des sichtbaren Bereichs zu liefern. Aus diesen beiden Punkten kann nun der für das Mousepicking benötigte Strahl berechnet werden.

Bei der Übergabe der Bildschirmkoordinaten ist jedoch zu beachten, dass das von Swing

verwendete Koordinatensystem seinen Ursprung in der *oberen* linken Ecke besitzt. Im Vergleich zu OpenGL, das den Ursprung mathematisch eingängig in der *unteren* linken Ecke platziert. Die y-Koordinate ist also vor der Berechnung von der Höhe des Bildschirms abzuziehen.

In Abbildung 4.13 ist eine Visualisierung der Strahlen nachzuvollziehen.

4.3.5 Bewegungskegel

Im der prototypischen Implementierung ist es möglich, eine Reichweitenanalyse zu einer Einheit durchzuführen. Das heißt, wie weit sie sich unter Annahme einer bestimmten Maximalgeschwindigkeit bewegen kann. Dargestellt wird diese Reichweite durch einen Kegel, der seine Spitze im Ausgangsort der Einheit hat.

Interessant ist es, festzustellen, welche anderen Einheiten sich in dem Kegel befinden. Visuell ist dies meist möglich, das heißt eine Berechnung ist nicht immer notwendig. Aber es gibt Grenzfälle, in welchen eine genaue Aussage nur mathematisch getroffen werden kann. Wie diese Berechnung aussehen kann, soll im Folgenden dargelegt werden.

Engine-Mittel

Unter Berücksichtigung der Tatsache, dass eine Engine für die dreidimensionale Darstellung verwendet wurde, kann man davon ausgehen, dass diese Mittel zur Verfügung stellt, um das Problem zu lösen.

Eine mögliche Umsetzung basiert auf der Technik, die auch für die Mousepicks (siehe Abschnitt 4.3.4) verwendet wurde: Gegeben seien die dreidimensionalen Objekte der Einheiten und der Kegel, in dem sie potentiell liegen könnten. Von jeder Einheit wird jetzt ein Strahl ausgesendet. Richtung ist grundsätzlich die negative y-Richtung. Naiv gesprochen: *nach unten*.

Genau wie beim Mousepicking wird jetzt geprüft, ob der Kegel getroffen wurde. War dies der Fall, befindet sich die Einheit darin.

In der prototypischen Implementierung wurde diese Variante nicht umgesetzt. Grund dafür ist sind Probleme mit dem Algorithmus, den die Engine für die Kollisionsberechnung nutzt. Er ist auf Geschwindigkeit, nicht jedoch auf Genauigkeit ausgelegt. Ziel der Berechnung sollte aber ein sehr genaues Ergebnis sein. Deswegen entfällt diese Variante.

Winkelvergleich

Ein Ansatz das Problem mit Analytischer Geometrie anzugehen, ist der Vergleich von Winkeln zwischen Vektoren. Für diese Variante kann o. B. d. A. der Kegel als (gleichschenkliges) Dreieck angenommen werden. Auf der Basis wird nun das Lot gefällt. Bei diesem handelt es sich um die Höhe des Dreiecks.

Als Erstes wird der Winkel zwischen den Schenkeln des Dreiecks und dem Lot berechnet. Da der Radius des Kegels mit der Höhe des Dreiecks (und ebenfalls der Höhe des Kegels) ein rechtwinkliges Dreieck aufspannt, gilt:

$$\tan \alpha = \frac{r}{h} \tag{4.3.8a}$$

$$\alpha = \arctan \frac{r}{h} \tag{4.3.8b}$$

Als Zweites wird der Winkel berechnet, der sich für jede zu prüfende Einheit zwischen der Strecke von der Spitze des Kegels zur Einheit und dem Lot befindet. Er berechnet sich nach:

$$\cos \beta = \frac{\vec{a} \bullet \vec{b}}{|\vec{a}| \cdot \left|\vec{b}\right|} \tag{4.3.9}$$

Ist dieser Winkel größer, liegt die Einheit außerhalb des Kegels. Ist er anderenfalls kleiner, befindet sich die geprüfte Einheit im Kegel.

Insgesamt handelt es sich hierbei um ein recht rechenaufwendiges Verfahren, da es für jede Einheit durchgeführt werden muss. Aus diesem Grund stellt sich die Frage, ob das es Möglichkeiten zur Vereinfachung gibt.

Vergleich der y-Komponenten Berechnet man den Winkel α nicht mit der Gleichung 4.3.8, sondern mit Gleichung 4.3.9, ergibt sich eine Vereinfachung: Da der Richtungsvektor nur in der y-Komponente von null verschieden ist, geht auch nur die y-Komponente des Schenkelvektors ein. Damit degeneriert das Skalarprodukt zu einer einzigen Multiplikation.

Für die Winkelbrechnung sind die Längen der Vektoren, die für die Berechnung herangezogen werden, unerheblich. Darum muss für das Lot nicht mit dem Vektor (0, h, 0) gerechnet werden. Es genügt (0, 1, 0). Das Skalarprodukt muss nun schlicht nicht mehr ausgerechnet werden.

Ebenso können der Schenkelvektor und der Vektor von der zu prüfenden Einheit Richtung Kegelspitze normalisiert werden. Resultat ist, dass der Nenner der Formel zur Winkelberechnung gleich eins ist. Die Berechnung des Arkuskosinus ist zum Vergleich der beiden Winkel hinfällig. Schließlich müsste er nur von den y-Komponenten berechnet werden. Diese können aber auch gleich verglichen werden: Ist die des normalisierten Vektors der zu prüfenden Einheit kleiner, liegt sie außerhalb. Ist die des normalisierten Schenkelvektors kleiner, liegt sie im Kegel.

Zusammenfassen müssen zur Berechnung nur noch folgende Schritte durchlaufen werden:

1. Einmalige Berechnung des Schenkelvektors
2. Normalisierung des Schenkelvektors
3. Für jede zu prüfende Einheit:
    a) Berechnung des Vektors von ihr zur Kegelspitze
    b) Normalisierung dieses Vektors
    c) Vergleich der y-Komponenten der beiden Vektoren.

Durch diese Optimierung werden pro zu prüfender Einheit folgende absoluten und relativen Einsparungen gemacht:

absolute Verbesserung	*relative Verbesserung*
-4 Additionen	-44%
-6 Multiplikationen	-66%
-3 Divisionen	-50%
-1 Wurzel	-50%
-1 trigonometrische Funktion	-100%

Tabelle 4.4: Erreichte Verbesserungen durch Vereinfachung der Berechnung

5 Fazit und Ausblick

5.1 Bewertung

5.2 Weiterführende Arbeit

5.3 Fazit

Literaturverzeichnis

[Bur09] Bureau, U. S. Census: *Geographic Information Systems FAQ, What is the best way to calculate the distance between 2 points?*, 2009. [Online; Stand 14. Dezember 2009].

[Dav05] Davison, Andrew: *Killer Game Programming in Java.* O'Reilly Media, Inc., May 2005.

[GHJV02] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design patterns: abstraction and reuse of object-oriented design*, Seiten 701–717. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[Hal92] Hall, David L.: *Mathematical Techniques in Multisensor Data Fusion.* Artech House, Inc., Norwood, MA, USA, 1992.

[HL97] Hall, D. L. und J. Llinas: *An introduction to multisensor data fusion.* Proceedings of the IEEE, 85(1):6–23, 1997.

[Sch08] Schregenberger, Ulrich: *Lernkurs zum Erstellen eines WYSIWYG-Editors in einer 3D Game Engine.* no Note, 2008.

[Var97] Varshney, P. K.: *Multisensor data fusion.* Electronics & Communication Engineering Journal, 9(6):245–253, 1997.

[Wik09a] Wikipedia: *Leopard 2 — Wikipedia, Die freie Enzyklopädie*, 2009. [Online; Stand 14. Dezember 2009].

[Wik09b] Wikipedia: *Truppenübungsplatz Bergen — Wikipedia, Die freie Enzyklopädie*, 2009. [Online; Stand 14. Dezember 2009].

Abbildungsverzeichnis

4.1 Übersicht des Datenmodells 18
4.2 Aufbau der Klasse Data 19
4.3 Eigenschaftssystem im Datenmodell 21
4.4 Übersicht des Frameworks (Frontend/Backend) 23
4.5 Ablauf des Visualisierungsprozesses 30
4.6 Darstellung einer Lage mit zwei aufeinandertreffenden Bataillonen, insgesamt 251 Datensätze 34
4.7 BattleSimVis als prototypische Implementierung einer militärischen Lagedarstellung 36
4.8 Baumstruktur der textuellen Darstellung 38
4.9 Filter mit voreingestellten Sichten auf die Lage 39
4.10 Hervorhebung in der grafischen Ansicht 40
4.11 Bewegungskegel in die Vergangenheit 41
4.12 Kugeldreieck mit Beschriftung 45
4.13 Darstellung der von der Maus ausgehenden Strahlen aus [Sch08] 48

Listings

4.1 Beispiel für eine wohlgeformte XML-Datei 24
4.2 Beispielhafte Implementierung der *extractDataFromNode()*-Methode . . . 26