Institut für Parallele und Verteilte Systeme
Abteilung Anwendersoftware
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 2716

Strombasierte Datenverarbeitung in einer verteilten Ausführungsumgebung

Carlos Lübbe

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. B. Mitschang
Betreuer:	Dipl.-Inf. Nazario Cipriani Dipl.-Inf. Mike Eissele (VIS)
begonnen am:	16. Januar 2008
beendet am:	17. Juli 2008
CR-Klassifikation:	C.2.4, H.2.4, H.5.1, I.3.2, I.3.5, I.3.6

Kurzfassung

Das NEXUS-Projekt hat das Ziel, eine offene, föderierte Plattform für kontextbezogene Anwendungen zu entwickeln. Grundlage hierfür bildet ein globales Kontextmodell, mit dessen Hilfe Anwendungen Zugriff auf Kontextinformationen erhalten können. Der Kontext einer Anwendung ist zum Beispiel deren Ausführungsort oder verschiedene Sensoren, die zum Ausführungszeitpunkt zur Verfügung stehen. Häufig besteht der Bedarf, die abstrakten Konstextinformationen, insbesondere von Sensoren, auf eine für den Menschen verständliche Art zu visualisieren. Gerade wegen der starken Zunahme von mobilen, mit Sensoren ausgestatteten Endgeräten, die Kommunikationsfähigkeiten besitzen, werden für eine Visualisierung von Daten flexible und verteilte Verarbeitungsstrukturen nötig. In einem solchen Umfeld sind Netwerkstrukturen häufig Fluktuationen unterworfen, was eine Anpassung an unterschiedlichste Situationen erforderlich macht. Aufgrund der hohen Flexibilität bei gleichzeitiger hoher Skalierbarkeit scheint ein Peer-to-Peer-Verbund als Netzwerkstruktur sinnvoll. Um sich an das häufig ändernde Hardware-Umfeld der Knoten anpassen zu können, sollte die Verarbeitung virtualisiert vorgenommen werden, das heißt, unabhängig von der unterliegenden Hardware oder den vorhandenen physischen Netzwerkverbindungen. Die dabei auftretenden Daten sind häufig von transienter Natur, so dass oft kein Nutzen aus einer persistenten Speicherung gezogen werden kann. Hier bietet sich die strombasierte Verarbeitung der Daten an.

In dieser Arbeit wird ein System zur Visualisierung von abstrakten Informationen in einem verteilten, virtualisierten Umfeld entwickelt. Die dabei anfallenden Daten werden auf strombasierte Weise verarbeitet. Als Plattform für die verteilte und virtualisierte Ausführung wird die NEXUS Datenstrom-Föderation [Sar08] verwendet.

Inhaltsverzeichnis

Kurzfassung **3**

1. Einleitung **9**
1.1. Anwendungsszenarien 10
1.2. Gliederung dieser Arbeit 11

2. Grundlagen **13**
2.1. Visualisierung 13
2.1.1. Visualisierungspipeline 14
2.2. Virtualisierung 17
2.3. Datenströme 18
2.3.1. Datenmodell 18
2.3.2. Operatoren 19
2.3.3. Techniken der näherungsweisen Anfragebeantwortung 20
2.3.4. Anfragearten 21
2.3.5. Data Stream Management Systeme (DSMS) 22
2.4. Nexus 22
2.4.1. Augmented World Model (AWM) 23
2.4.2. Datenstrom-Föderation 26
2.5. Forschungsstand 31

3. Entwurf **35**
3.1. Anforderungen 35
3.2. Entwurfsdiskussion 36
3.2.1. Interaktion 37
3.2.2. Datenmodell und Datenformat der ausgetauschten Objekte 39
3.2.3. Koordinatensysteme der Bildgenerierung 40
3.2.4. Geographische Koordinatentransformation 41
3.2.5. Zustand der Operatoren 43
3.2.6. Typisierung von Datenströmen 47
3.2.7. Modellierungssprache 50
3.2.8. Anfragesprachen 55

4. Technologie **59**
4.1. JOGL . . . 59
4.2. Java3D . . . 60
4.3. Quicktime-Api . . . 61
4.4. JMF . . . 62

5. Implementierung **65**
5.1. Komponenten . . . 65
5.1.1. CoordTransform-Operator . . . 65
5.1.2. NAVGInterpolator-Operator . . . 67
5.1.3. ExpressionMapper-Operator . . . 69
5.1.4. Bildgenerierungs-Komponenten . . . 70
5.1.5. Transport-Komponenten . . . 73
5.1.6. Videostrom-Verbindung zum Client . . . 74
5.1.7. Funktionsweise . . . 76
5.2. Evaluation . . . 76
5.2.1. Interpolation mit dem NAVGInterpolator . . . 77
5.2.2. Bildrate des Senders . . . 79
5.2.3. Instanzierungsdauer von VisualObjects . . . 81
5.2.4. Serialisierung und Deserialisierung . . . 82
5.2.5. Verarbeitungszeit von Kommandos . . . 83
5.2.6. Benötigte Bandbreite der Netzwerkverbindung . . . 84
5.2.7. Bildrate des Empfängers . . . 85
5.2.8. Latenz der Interaktion . . . 86

6. Visualisierungs-Beispiele **91**
6.1. Visualisierung von mobilen Objekten . . . 91
6.1.1. Problem . . . 91
6.1.2. Lösung . . . 92
6.2. Höhenfeld . . . 94
6.2.1. Triangulationsverfahren . . . 94
6.2.2. Problem . . . 97
6.2.3. Lösung . . . 97

7. Zusammenfassung **101**
7.1. Schlussfolgerung . . . 101
7.2. Entwicklungsmöglichkeiten . . . 102
7.2.1. Nexus-Datenmodell . . . 102
7.2.2. Erweiterung um die DSMS-Funktionalität . . . 103
7.2.3. Binärrepräsentation von GenericObjects . . . 103
7.2.4. Entwicklung eines blockierenden Sende-Mechanismus . . . 104
7.2.5. Weiterentwicklung der Datenstrom-Föderation zu einer Medien-Plattform . . . 105

Anhang **107**

A. VisualClassSchema **109**

B. VisualAttributeSchema **119**

Abbildungsverzeichnis **127**

Tabellenverzeichnis **129**

Literaturverzeichnis **131**

Kapitel 1

Einleitung

Technologischer Fortschritt im Bereich der Mikroeletronik und Kommunikationstechnik führen zunehmend zu einem stark vernetzten, mit Sensoren ausgestatteten Umfeld. Vertreter des Ubiquitious Computing [Mat03] prophezeien eine allmähliche Informatisierung des Alltags, indem Alltagsgegenstände mit Mikroprozessoren und Sensoren ausgestattet werden und miteinander kommunizieren. Der heute noch überall präsente Computer wird als sichtbares Gerät verschwinden, aber gleichzeitig allgegenwärtig in unserer Umwelt integriert sein. In einem solchen Umfeld sind viele Anwendungen denkbar, die abhängig von einem gewissen Kontext reagieren. Das Projekt NEXUS beschäftigt sich mit der Schaffung einer Plattform für kontextbezogene Anwendungen. Um die speziellen Anforderungen von Anwendungen, welche Informationen aus ihrem Umfeld durch eine Vielzahl von Sensoren gewinnen, zu berücksichtigen, wurde mit der NEXUS *Datenstrom-Föderation (Stream Federation)* eine Plattform geschaffen, welche die verteilte, strombasierte Verarbeitung von Daten zulässt [Sar08].

Neben der Zunahme von Mikroprozessoren und Sensoren in Alltagsgegenständen ist auch eine Zunahme von graphischen Ausgabemöglichkeiten zu erwarten. Mobiltelefone und Personal Digital Assistants (PDAs) mit Flüssigkristallbildschirmen (Liquid Crystal Displays, LCDs) sind Beispiele, die diese Entwicklung schon heute sichtbar machen. Es ist durchaus wahrscheinlich, dass diese Entwicklung noch weiter in unseren Alltag vorstoßen wird. Durch die Entwicklung von elektronischem Papier (E-Paper) ist es möglich, nahezu jede Oberfläche mit der Fähigkeit zu versehen und bestimmte Dinge graphisch wiederzugeben [Coo08]. So sind beispielsweise elektronische Tapeten denkbar, die neben ihrer ästhetischen Funktion gleichzeitig auch als Anzeigegerät fungieren. Gerade in Gegenwart von abstrakten Sensorinformationen ist es häufig sinnvoll, diese auf eine für den Menschen verständliche Art zu visualisieren. Die Visualisierung von Daten in einem solchen Umfeld ist hochgradig dynamisch und kontextabhängig. So könnte zum Beispiel die aktuelle Position einer Person in einem Raum mit elektronischer Tapete bestimmen, an welchem Ort die Sensorinformationen visualisiert werden. Ändert die Person ihren momentanen Aufenthaltsort, so wäre es wünschenswert, dass sich das Visualisierungssystem an die neue Situation anpasst.

Jedoch kann man von „smarten“ Alltagsgegenständen auch in einer digitalisierten Welt nicht die Rechenleistung eines Super-Computers erwarten. Die Leistungsfähigkeiten der

Gegenstände wird an das aktuelle Umfeld und die Ressourcen, die dort zur Verfügung stehen, wie z.B. Energie, angepasst sein. Dadurch entsteht der Bedarf, die komplexen, speicher- und leistungsintensiven Aufgaben der Visualisierung nicht auf dem Gerät durchzuführen, welches später für die Ausgabe der Visualisierung zuständig ist. Vielmehr wird in dieser Arbeit eine Verteilung der für die Visualisierung von Daten benötigten Schritte auf mehrere Rechner innerhalb eines Netzwerkes angestrebt. Die Daten, die bei den einzelnen Verarbeitungsschritten der Visualisierung entstehen, werden zwischen den Rechnern über ein Netzwerk ausgetauscht. Zwar fallen bei einer Visualisierung unter Umständen riesige Datenmengen an, die übertragen werden müssen, jedoch ist dieser Ansatz nicht ganz abwegig. Denn nach dem Gesetz von Edholm [Che04] wachsen die Übertragungsraten verschiedener Netzwerktechnologien in exponentieller Weise. Die heute zur Verfügung stehenden Übertragungsraten lassen schon jetzt die verteilte Visualisierung recht komplexer Szenarien zu und aufgrund technischer Weiterentwicklungen werden in Zukunft immer komplexere Szenarien visualisiert werden können.

In der hochgradig dynamischen Welt des vorausgesagten Ubiquitious Computing kann nicht von festen relativ statischen Netzwerken, wie sie heute vorliegen, ausgegangen werden. Die mit Mikroprozessoren versehenen Alltagsgegenstände werden abhängig von ihrer Position und ihren Fähigkeiten spontane Verbindungen mit anderen Alltagsgegenständen aufnehmen, um gewisse Aufgaben zu realisieren oder Dienste anzubieten. Ein an einem bestimmten Ort angebotener Dienst muss auf unterschiedlichster Hardware ausgeführt werden können. Möglicherweise wird es sogar nötig, während der Berechnung einer Aufgabe oder der Realisierung eines Dienstes, operative Teile auf andere Hardware zu migrieren. Die operative Teile sind somit *virtualisiert*, das heißt, unabhängig von der ausführenden Hardware-Umgebung.

Das Ziel dieser Arbeit ist es, ein System zur verteilten, virtualisierten Visualisierung von abstrakten Sensorinformationen zu entwickeln. Der Visualisierungsprozess selbst lässt sich in gewisse Teilschritte unterteilen und in einer so genannten *Visualisierungspipeline* anordnen. Diese Teilschritte sollen in einem verteilten Umfeld ausgeführt werden. Die Ausführung kann dabei parallel erfolgen und die Ergebnisse der einzelnen Teilschritte sollen dabei auf eine strombasierte Weise verarbeitet werden. Die benötigte Flexibilität für die Anpassung an das dynamische Umfeld soll durch eine Virtualisierung der operativen Teile erreicht werden. Als Plattform für die verteilte und virtualisierte Ausführung der operativen Teile des Systems wird die NEXUS Datenstrom-Föderation nach [Sar08] verwendet.

1.1. Anwendungsszenarien

Das Spektrum von Anwendungen, die in einem solchen Umfeld möglich oder denkbar sind, ist enorm. Im Folgenden sollen beispielhaft einige Anwendungsszenarien angeführt werden.

1. Ein Benutzer möchte, dass die eigene Position und die seiner Freunde, die maximal 500 Meter entfernt sind, in einer Karte auf seinem Mobiltelefon angezeigt wird (vgl. [Sar08]).
2. Im Umfeld einer „Smart Factory“ [BJRW03] möchte ein Arbeiter die Position und den Zustand eines mit Sensoren bestückten Werkzeuges erfahren. Die Werkzeuge könnten zum Beispiel als farbige Punkte innerhalb eines Lageplans auf einem mobilen Endgerät, wie einem PDA, visualisiert werden. Die jeweilige Farbe beschreibt dann den Zustand des Gerätes. Zum Beispiel könnte die Farbe rot ausdrücken, dass das Wartungsintervall des Werkzeuges überschritten wurde und das Werkzeug gewartet werden muss, während die Farbe grün ein funktionsfähiges, gewartetes Werkzeug anzeigt.
3. Ein Pilot eines Passagierflugzeuges sieht während des Fluges eine von seiner Flugposition abhängige Wetterkarte auf einem Bildschirm. Die Wetterkarte wird aus Sensorinformationen berechnet, welche von am Boden stationierten Wetterstationen ausgesendet werden. Die Wetterkarte ist „live“ und somit kann der Pilot lokalen Unwettern gezielt ausweichen.
4. Die Positionen von mit GPS bestückten Automobilen werden in einem dreidimensionalen virtuellen Stadt-Panorama visualisiert. Dies kann zum Beispiel bei Verkehrsuntersuchungen den Wiedererkennungswert gewisser Problemzonen steigern. Die Visualisierung von mobilen Objekten wird in Kapitel 6.1 vorgenommen.
5. Die Signalstärken eines Mobilfunknetzes könnten zum Beispiel mit Hilfe eines so genannten *Höhenfeldes* visualisiert werden. Die Visualisierung von Höhenfeldern wird im Kapitel 6.2 behandelt.

1.2. Gliederung dieser Arbeit

Die Motivation und das Ziel dieser Arbeit wurden kurz in diesem Kapitel beschrieben. In Kapitel 2 werden die Grundlagen der Visualisierung, der strombasierten Datenverarbeitung und die grundlegenden Begriffe und Entitäten der NEXUS Datenstrom-Föderation erläutert. Außerdem wird ein Vergleich zu ähnlichen Forschungsprojekten oder -gebieten gezogen. Kapitel 3 beschreibt den Grobentwurf der in dieser Arbeit realisierten Visualisierungspipeline. Die hierfür verwendbaren Technologien werden in Kapitel 4 bewertet. Eine detaillierte Beschreibung der Implementierung des Prototyps ist in Kapitel 5 zu finden. Beispiele einer Visualisierung für die letzten beiden Anwendungsszenarien (siehe 1.1) finden sich in Kapitel 6. Letztlich wird die Arbeit in Kapitel 7 bewertet und mögliche Felder der Weiterentwicklung des Systems geschildert.

Kapitel 2

Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe und Konzepte der Visualierung, Virtualisierung, der strombasierten Datenverarbeitung sowie der NEXUS Datenstrom-Föderation vorgestellt.

2.1. Visualisierung

Der Mensch nutzt seit jeher Bilder oder bildhafte Darstellungen dazu, um Informationen zu präsentieren. Ein sehr frühes Beispiel hierfür sind Höhlenmalereien aus der Altsteinzeit. Auch in der heutigen Zeit und vor allem im Bereich der elektronischen Datenverarbeitung ist es immer noch üblich, gewisse Fakten oder Zusammenhänge graphisch darzustellen, insbesondere wenn es sich um große Datenmengen handelt. Der Prozess zur Erstellung solcher bildhaften Darstellungen wird *Visualisierung* genannt. Nach [McC87] hat die Visualisierung das Ziel, *„das Symbolische in das Geometrische"* zu überführen. Dabei geht es vor allem darum, abstrakte Daten zu präsentieren, um so das Verständnis der Daten zu fördern und dabei eventuell innere, sonst verborgene Zusammenhänge aufzuzeigen. Eine andere Definition schließt damit nicht nur visuelle Darstellungen mit ein: Nach [FR94] ist die Visualisierung die Abbildung von Daten auf Repräsentationen, die wahrgenommen werden können. Dies schließt zum Beispiel die auditive oder taktile Wahrnehmung mit ein. Neben der Verständnisförderung wird die Visualisierung häufig zur Erleichterung der Kommunikation von Informationen verwendet.

Damit eine Visualisierung von abstrakten Daten den gewünschten Erfolg, wie zum Beispiel Verständnisförderung erbringt, sollte sie *angemessen*, möglichst *effektiv* und außerdem *expressiv* sein (siehe [SM00]). Die *Expressivität* bezeichnet die Ausdrucksfähigkeit einer visuellen Darstellung. Die Informationen, die in der visuellen Darstellung enthalten sind, sollten sich nicht von den Informationen, die in den visualisierten Daten vorhanden sind, unterscheiden. Suggeriert eine visuelle Darstellung Informationen, die nicht in den Daten vorhanden sind, so ist das Expressivitätskriterium verletzt. Die *Effektivität* bezeichnet die Optimalität einer visuellen Darstellung bezüglich der Fähigkeiten des Betrachters unter Berücksichtigung der Eigenschaften des Ausgabegerätes und des Anwendungskontextes. Die Effektivität kann

zum Beispiel durch die Zeit gemessen werden, die ein Betrachter benötigt, um visuelle Darstellungen zu verstehen. Während die Effektivität den Nutzen einer visuellen Darstellung betrachtet, betrachtet die *Angemessenheit* die Kosten in Relation zu dem Nutzen. Wenn es zum Beispiel eine Visualisierung von Daten gibt, die ähnlich effektiv ist, aber wesentlich weniger Ressourcen zur Berechnung der Visualisierung benötigt, so ist diese vorzuziehen.

Nach [SM00] kann der Ursprung der zu visualisierenden Daten entweder in der realen Welt, der theoretischen Welt oder der künstlichen Welt liegen. Erstere werden zum Beispiel durch Sensormessungen gewonnen, zweitere auf der Grundlage von mathematischen Modellen berechnet und letztere werden entworfen, wie zum Beispiel Gemälde, Fernsehen oder Filme. In der Visualisierung wird jedoch in der Regel von dem Ursprung der Daten abstrahiert. Es werden lediglich die Eigenschaften des so genannten *Beobachtungsraumes* klassifiziert. Eine dieser Eigenschaften ist zum Beispiel die *Dimensionalität* des Beobachtungsraumes. Wurden die Daten in der realen Welt erhoben, so könnte der Beobachtungsraum zum Beispiel zwei- oder dreidimensional sein, während Daten aus der theoretischen Welt auch in einem theoretischen Raum mit beliebig vielen Dimensionen liegen können. Findet die Ausgabe auf einem zweidimensionalen Ausgabegerät statt, so muss eine Projektion durchgeführt werden. Hierbei kann es zu Verdeckungen oder Mehrdeutigkeiten kommen, die das Effektivitäts- oder Expressivitätskriterium verletzen. Eine weitere Eigenschaft des Beobachtungsraumes ist der *Wirkungskreis* eines Beobachtungspunktes. Dieser kann *punktuell*, *lokal* oder *global* sein. Dabei gelten die Daten eines Punktes entweder nur für den zugehörigen Beobachtungspunkt, für eine gewisse Umgebung des Beobachtungspunktes oder für den gesamten Beobachtungsraum. Die letzte Eigenschaft des Beobachtungsraumes ist die *Topologie* der beobachteten Daten. Die Topologie definiert den Zusammenhang oder die Konnektivität der Daten. Werden zum Beispiel unabhängige Sensorwerte an verschiedenen Stellen des Beobachtungsraumes genommen, wird man häufig von *zerstreuten Daten (scattered data)* sprechen. Es besteht in diesem Fall keine Konnektivität zwischen einzelnen Punkten des Beobachtungsraumes. In anderen Fällen ist aber eine Konnektivität zwischen den Daten vorhanden, bzw. wird explizit während des Visualisierungsprozesses hergestellt.[1] Die verschiedenen vorkommenden Topologien werden oft in verschiedene Gittertypen klassifiziert. Hierfür sei zum Beispiel auf [SM00] verwiesen.

2.1.1. Visualisierungspipeline

Die für den Visualisierungsprozess benötigten Schritte lassen sich auf eine bestimmte Weise anordnen (vgl. hierzu [SM00]). Die so genannte *Visualisierungspipeline* setzt sich aus drei wesentlichen Schritten zusammen: die *Datenaufbereitung (Filtering)*, die *Geometrieerzeugung (Mapping)* und die *Bildgenerierung (Rendering)*. Abbildung 2.1 zeigt eine solche Visualisierungspipeline.

Die *Datenaufbereitung* realisiert eine Abbildung von Daten zu Daten. Anfangs erhebt eine Anwendung, wie beispielsweise eine Simulation, eine Vielzahl von Daten, die so genannten *Rohdaten*. Möglicherweise befinden sich in dieser Datenmenge einige Daten, die für die

[1] Die in Kapitel 6.2 beschriebene Triangulierung ist ein Beispiel hierfür.

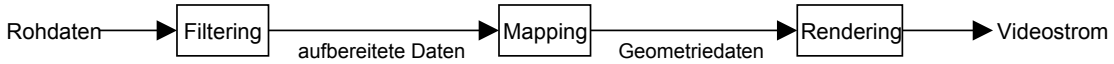


Abbildung 2.1.: Stufen der Visualisierungspipeline

aktuelle Visualisierung nicht von Belang sind. Die Auswahl der benötigten Daten charakterisiert die Aufgabe der *Datenauswahl* oder auch *Selektion*. Weiterhin ist es möglich, dass die ausgewählten Daten nicht für eine stufenlose, sinnvolle Visualisierung ausreichen. In diesem Fall müssen fehlende Datenwerte durch *Interpolation* ergänzt werden. Letztendlich lässt sich die *Filterung* von Daten als Teilbereich der Datenaufbereitung angeben. Im Falle von Daten, welche aus einer Messung der Umwelt mit Hilfe von Sensoren gewonnen wurden, verfälschen häufig Messfehler das Ergebnis. Diese können zum Beispiel durch Schwellwertoperationen geglättet werden. Gelegentlich ist es auch von Nutzen, gewisse Eigenschaften der Daten, zum Beispiel durch Berechnung der Extremstellen, besonders hervorzuheben. Oftmals werden die Begriffe *Interpolation* und *Filterung* unter dem Begriff *Anreicherung (Enhancement)* zusammengefasst.

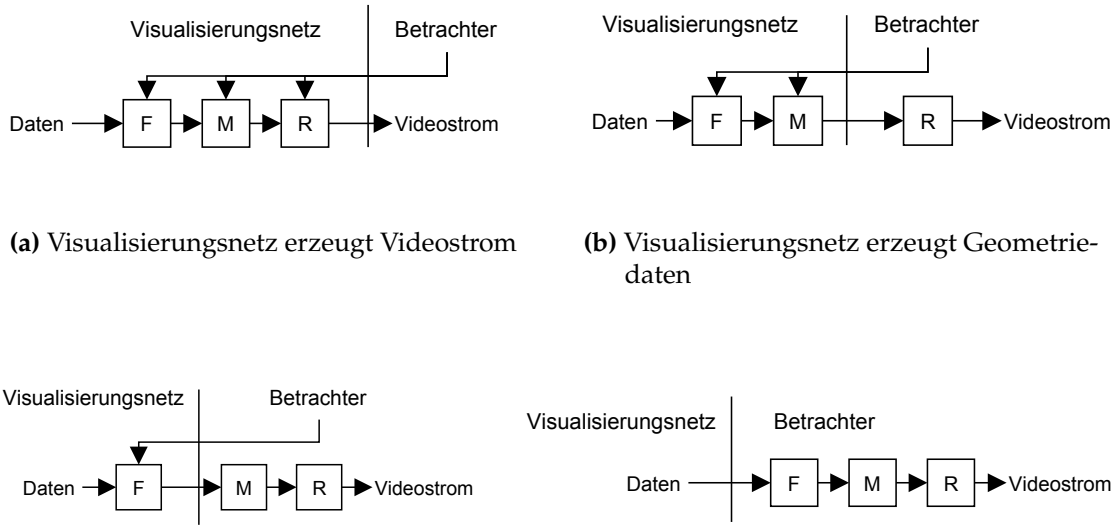
Die *Geometrieerzeugung* zielt darauf ab, die aufbereiteten Daten in geometrische Daten zu überführen. Während dieses Schrittes wird festgelegt, welche geometrischen Primitive erzeugt werden und mit welchen Attributen, wie zum Beispiel die Farbe, sie versehen werden. Damit ist die Geometrieerzeugung eine Abbildung von beliebigen Daten auf Geometriedaten. Die Geometriedaten können sich stark von den Daten unterscheiden, aus denen sie hervorgegangen sind. Beispielsweise könnte der Datenpunkt einer Position auf das komplexe, geometrische Modell eines Automobils abgebildet werden. Die Geometriedaten sind abstrakt, das heißt, sie beschreiben die darzustellende Geometrie.[2]

Während des *Bildgenerierungs*-Schrittes werden aus den Geometriedaten Bilder, bzw. eine Sequenz von Bildern, generiert. Die Beschreibung der Algorithmen und Methoden der Bildgenerierung würde den Rahmen dieser Arbeit überschreiten. Für die Bildgenerierung existieren viele Lösungen und Technologien, die für diese Arbeit verwendet werden können (siehe Kapitel 4).

Die oben beschriebene Visualisierungspipeline muss nicht als ein einziger Prozess realisiert werden. Nach [WBW96] können die einzelnen Schritte auch auf mehrere Rechner verteilt werden. Dies hat den Vorteil, dass ein einzelner Rechner nicht alle, mitunter sehr komplexen Schritte der Visualisierungspipeline durchführen muss. Die Gesamtlast kann dann auf mehrere Rechner verteilt werden. Dazu können eventuell einzelne Schritte parallelisiert werden, um so den Gesamtdurchsatz zu erhöhen. Außerdem hat der Bildgenerierungs-Schritt häufig spezielle Anforderungen an die Hardware eines Systems. In einem verteilten Szenario ist es möglich, die beteiligten Rechner für ihre spezielle Aufgabe innerhalb der Visualisierungspipeline optimal zu konfigurieren, bzw. gegebenenfalls mit spezieller Hardware auszustatten. Es stellt sich die Frage, welche Schritte von dem Netzwerk der unabhängigen

[2] Hier werden wie auch in [SM00] die Begriffe „Geometrieerzeugung“ und „Geometriedaten“ verwendet. „Geometrie“ ist hier eher im Sinne von darstellbarer Repräsentation zu verstehen und das schließt zum Beispiel Text mit ein.

Rechner und welche von dem Betrachter durchgeführt werden. [WBW96] hat hier einige Szenarien untersucht. Jedoch wird hier von einem Request-Response-Ansatz ausgegangen, bei dem ein *Betrachter (Viewer)* Anfragen an einen *Verteiler (Publisher)* stellt. Der Verteiler ist ein Server, der seine Funktionalität über das World Wide Web anbietet. Die von [WBW96] getroffene Klassifizierung wird in dieser Arbeit nicht auf das Request-Response-Paradigma beschränkt. Die Funktionalität des *Verteilers* kann auf mehrere Knoten innerhalb eines Netzwerkes verteilt sein. Das Ergebnis einer Visualisierungsanfrage muss nicht unbedingt ein einzelnes Bild sein, sondern kann auch einen Videostrom darstellen, der über die Zeit hin gesendet wird. Die einzelnen Knoten des Netzwerkes verarbeiten die ankommenden Daten auf strombasierte Weise (siehe hierzu Kapitel 2.3). Um dies zu verdeutlichen, wird im Folgenden auf den Begriff *Verteiler* verzichtet und stattdessen der Begriff *Visualisierungsnetz* eingeführt. Abbildung 2.2 zeigt die für diese Arbeit in Erwägung gezogenene Möglichkeiten der Aufteilung. Es wird davon ausgegangen, dass der Betrachter die Möglichkeit hat, auf alle Schritte der Visualisierungspipeline Einfluss zu nehmen. Falls diese Teilschritte von dem Visualisierungsnetz ausgeführt werden, so geschieht dies durch setzen bzw. aktualisieren von Parametern der einzelnen Komponenten. Werden dabei Rechnergrenzen überschritten, so ist dies mit einem Pfeil vom Betrachter zur jeweiligen Komponente angezeigt. Werden die Komponenten vom Betrachter ausgeführt, so besteht in jedem Fall unmittelbarer Zugriff auf Komponenten, weshalb dies in der Abbildung nicht explizit verdeutlicht wird.



(a) Visualisierungsnetz erzeugt Videostrom

(b) Visualisierungsnetz erzeugt Geometriedaten

(c) Visualisierungsnetz bereitet Daten auf

(d) Betrachter erzeugt die Visualisierung

Abbildung 2.2.: Möglichkeiten der Verteilung einer Visualisierungspipeline. „F“ steht für Filtering, „M“ für Mapping und „R“ für Rendering

Bei der ersten Aufteilung (2.2a) werden alle Teilschritte der Visualisierungspipeline vom Netzwerk durchgeführt. Dies hat den Vorteil, dass für die einzelnen Schritte der Visualisierungspipeline spezialisierte Hardware eingesetzt werden kann. Weiterhin bedarf es nur einer leichtgewichtigen Betrachter-Komponente. Sie muss lediglich dazu fähig sein, einen Videostrom wiederzugeben und kann auf komplexe Hard- und Software verzichten, die beispielsweise für die Bildgenerierung benötigt wird.

Dies ist bei der zweiten Aufteilung nicht der Fall (2.2b). Hier erzeugt der Betrachter den Videostrom, dass heißt, der Betrachter muss die notwendige Hard- und Software mit sich bringen, um die Bildgenerierung durchführen zu können. Diese Aufteilungsmöglichkeit hat den Vorteil, dass der Betrachter unmittelbar die Parameter der Bildgenerierung ändern kann. Dies ermöglicht zum Beispiel ein flüssiges Navigieren durch die dreidimensionale Welt des Visualisierungsszenarios. Werden für die Parameteränderungen Rechnergrenzen überschritten, so muss mit einer Reaktionszeit ähnlich der im World Wide Web, also im Sekundenbereich, gerechnet werden.[3]

Bei der dritten Aufteilung (Abbildung 2.2c) werden die Daten vom Visualisierungsnetz aufbereitet. Das ist zum Beispiel sinnvoll, wenn die Menge der zu verarbeitenden Daten für den Betrachter zu groß wäre, oder wenn komplexe Interpolationen durchgeführt werden müssen, wozu der Betrachter nicht in der Lage ist. Der Betrachter kann außerdem unmittelbar Einfluss auf den Mapping-Schritt nehmen. Damit kann sich beispielsweise die Farbe gewisser Objekte dynamisch, abhängig dem Zustand eines Eingabegerätes, wie der Maus, geändert werden.

Die letzte Aufteilungsmöglichkeit (Abbildung 2.2d) geht davon aus, dass alle Schritte der Visualisierungspipeline vom Betrachter durchgeführt werden. Damit wird ein Höchstmaß an Leistungsfähigkeit vom Endgerät des Betrachter gefordert. Anderseits wird dadurch auch ein Höchstmaß an Interaktion ermöglicht.

Prinzipiell sind alle oben beschriebenen Aufteilungen mit der Datenstrom-Föderation nach [Sar08] möglich. Die Aufteilungen können sogar dynamisch, das heißt, abhängig von den Wünschen des Betrachters oder den Fähigkeiten seines Endgerätes gewählt werden. Letztlich lässt sich feststellen, dass sowohl die Interaktivität als auch die Anforderungen an das Endgerät des Betrachters in der genannten Reihenfolge der Aufteilungen zunimmt.

2.2. Virtualisierung

Wie im vorherigen Abschnitt angedeutet ist es möglich, eine Visualisierungspipeline in einem verteilten Umfeld zu realisieren. Darüber hinaus ist es Ziel dieser Arbeit, die operativen Teile der Visualisierungspipeline *virtualisiert* zur Ausführung zu bringen. Der Begriff *„Virtualisierung“* ist breit gefächert, wird aber vor allem dann verwendet, wenn eine logische Gruppierung von bestimmten physischen Ressourcen oder Strukturen stattfindet, oder wenn von gewissen Geräten oder fest installierten Software-Komponenten abstrahiert wird. Die

[3] Siehe hierzu auch die Entwurfsdiskussion zum Thema „Interaktion“ im Kapitel 3.2.1.

Java Virtual Machine (JVM) zum Beispiel abstrahiert von verschiedenen Betriebssystemen und bietet somit eine virtuelle Laufzeitumgebung für alle Java-Anwendungen.

Wenn in dieser Arbeit von *Virtualisierung* gesprochen wird, so meint dies in der Regel die folgenden beiden Aspekte:

1. Die logischen Verarbeitungsschritte einer Visualisierungspipeline können von verschiedenen physischen Implementierungen der operativen Teile realisiert werden.
2. Die logische Anordnung der Verarbeitungsschritte ist unabhängig von der physischen Anordnung der Verarbeitungsknoten.

Andere Aspekte der Virtualisierung wie Plattform- oder Programmiersprachenunabhängigkeit sind zwar Teil der grundlegenden Idee der Datenstrom-Föderation, haben aber für diese Arbeit nur am Rande eine Bedeutung.

2.3. Datenströme

Die Visualisierungspipeline wird von Komponenten realisiert, die Daten in strombasierter Weise verarbeiten. Die strombasierte Datenverarbeitung ist seit geraumer Zeit ein Forschungsgebiet der Datenbank-Community. In diesem Abschnitt sollen die wesentlichen Begriffe und Modelle erläutert werden. Für vertiefende Informationen sei der interessierte Leser auf die Literatur (z.B. [BBD+02]) verwiesen.

2.3.1. Datenmodell

Das in der Datenbank-Literatur (z.B. [KE01]) vertretene Datenmodell, das so genannte *relationale Modell*, basiert auf Tabellen (Relationen) mit Zeilen und Spalten. Formal ist eine *Relation* eine Teilmenge des Kreuzproduktes $W_1 \times W_2 \times \ldots \times W_n$, wobei die Wertebereiche W_i in der Regel als atomar angesehen werden, das heißt, nicht aus zusammengesetzten Werten bestehen dürfen. Die Zeilen der Tabellen werden auch *Tupel* genannt, die Spalten werden häufig nur *Attribute* genannt. Man kann an dieser Stelle noch zwischen dem Namen der Spalte, dem *Attributnamen* und den Werten der Komponenten der Tupel, den *Attributwerten* differenzieren. Das relationale Modell ist ein flaches Datenmodell. Um Hierarchien zu modellieren, wird ein Referenzierungsmechanismus benötigt. Dieser muss explizit modelliert werden. Hierfür beinhaltet jedes Tupel einer Relation ein identifizierendes Attribut, den *Primärschlüssel*. Wird von einem Tupel T_1 einer Relation auf ein Tupel T_2 einer anderen Relation referenziert, so wird der Primärschlüssel von T_2 explizit als Attributwert eines Attributes, dem *Fremdschlüssel*, innerhalb von T_1 angegeben.

Für das relationale Modell wurden Operatoren spezifiziert, welche im Folgenden erläutert werden. Für die formale Definition sei auf die Literatur [KE01] verwiesen. Die *Selektion* $\sigma_P(R)$ wählt die Tupel aus R aus, die dem Prädikat P entsprechen. Die *Projektion* $\Pi_{A_1 \cdots A_k}(R)$ bildet eine neue Tabelle aus den Attributen $A_1 \cdots A_k$, während die nicht angegebenen Attribute verworfen werden. Der *Join* $R_1 \bowtie_P R_2$ ist das Kreuzprodukt zweier Relationen R_1 und R_2

mit anschließender Selektion über dem Prädikat P. Für eine sinnvolle Implementierung dieser Operatoren wird häufig angenommen, dass auf die Tupel einer Relation wahlfreier Zugriff besteht.

Das relationale Modell ist für die strombasierte Datenverarbeitung nur begrenzt anwendbar. Nach [BBD+02] besteht hier in der Regel kein wahlfreier Zugriff auf die Daten. Tupel werden als einzelne Informationseinheiten übertragen. Die Reihenfolge, in der die Elemente eines Datenstroms ankommen können, lässt sich im Allgemeinen weder vorhersagen, noch kontrollieren. Die Anzahl der Elemente, welche über die Zeit in einem Datenstrom übermittelt werden, ist normalerweise unbekannt und muss deshalb in vielen Fällen als unbeschränkt angenommen werden. Strombasierte Operatoren müssen ein Element eines Datenstroms bei seiner Ankunft entweder direkt verarbeiten und anschließend verwerfen, oder für seine spätere Verarbeitung zwischenspeichern.

Aufgrund des Mangels an wahlfreiem Zugriff stellt sich die beim relationalen Modell übliche Methode der Referenzierung mithilfe von Fremdschlüssel-Beziehungen als schwierig heraus. Außerdem wird es unmöglich, komplexe Attribute zu modellieren. Im relationalen Modell können diese nur durch weitere Relationen modelliert werden, welche aber in der Regel nicht verfügbar sind, da kein wahlfreier Zugriff besteht. Weiterhin treten speziell bei der Visualisierung Daten auf, die in binärer Repräsentation vorliegen, wie z. B. eine Rastergrafik, und deshalb nicht sinnvoll in Tupel und Attribute strukturiert werden können.

Dennoch ist das relationale Datenmodell nicht vollständig aus der strombasierten Welt zu verbannen. Statische Daten könnten in einem relationalen Format in einer Datenbank gespeichert sein, während dynamische Daten aus einem Datenstrom mit ihnen auf irgendeine Weise verknüpft werden. Bei der Visualisierung eines Automobils könnten die statischen Daten des geometrischen Modells, welche sich bei hoffentlich unfallfreier Fahrt über die Zeit hin nicht ändern werden, mit der dynamischen Positionsinformation eines GPS-Gerätes verknüpft werden.

Letzten Endes kommt es auf die Anwendungsdomäne an, welches Datenmodell als am sinnvollsten erscheint. Deswegen schlägt [Sar08] eine Plattform für die strombasierte Datenverarbeitung vor, welche kein spezielles Datenmodell postuliert. Informationen zwischen Operatoren werden in Form von abgeschlossenen Informationseinheiten, im Folgenden *Objekte* genannt, ausgetauscht. Dabei legt der Anwendungsentwickler das Format der Objekte selbst fest. Das ausgetauschte Objekt könnte zum Beispiel auch eine Relation sein, bestehend aus vielen Tupeln. Damit besteht also weiterhin die Möglichkeit für einzelne Operatoren auf einem relationalen Datenmodell zu operieren.

2.3.2. Operatoren

Operatoren lassen sich in *zustandsbehaftete* und *zustandslose* Operatoren klassifizieren. Zustandsbehaftete Operatoren müssen für die Verarbeitung Objekte zwischenspeichern, während zustandslose Operatoren ohne dies auskommen. Für Operatoren, die auf Mengen von Objekten arbeiten, wird häufig wahlfreier Zugriff auf die Objekte benötigt. Bei der strombasierten Datenverarbeitung können nun über die Zeit potentiell unbeschränkte Folgen von

Objekten auftreten. Würde man einen solchen Operator mit einer dieser unbeschränkten Folgen konfrontieren, so könnte der Operator kein Ergebnis produzieren, denn Objekte, die für die Verarbeitung benötigt werden, sind eventuell noch nicht bei dem Operator angekommen. Man sagt, der Operator würde *blockieren*. Anderseits gibt es auch Operatoren, die zwar für die Verarbeitung eines Objektes keine Objekte aus der Zukunft benötigen, jedoch Objekte der Vergangenheit. Um diese speichern zu können, bräuchte ein solcher Operator einen unendlich großen Speicher. Da jedoch in der Realität keine unendlich großen Speicher existieren, müssen andere Wege gesucht werden. Für viele der auf Mengen operierenden relationalen Operatoren lassen sich einfach Implementierungen finden, die auch in Gegenwart von unbeschränkten Objekt-Folgen nicht blockieren. Ein Beispiel hierfür ist die Selektion $\sigma_P(R)$. Das Prädikat P kann iterativ auf die ankommenden Objekte angewendet werden. Objekte, die dem Prädikat entsprechen, werden weitergeleitet, die anderen verworfen. Für andere Operatoren, wie der Join-Operator, können keine Implementierungen gefunden werden, die nicht blockieren oder unbeschränkten Speicher benötigen würden. Deswegen müssen Techniken der näherungsweisen Anfrageverarbeitung verwendet werden.

2.3.3. Techniken der näherungsweisen Anfragebeantwortung

Eine Möglichkeit hierfür ist es, Anfragen nicht über dem gesamten Datenstrom auszuwerten, sondern nur eine gewisse Menge der zuletzt angekommenen Elemente auszuwählen und die Anfrage über diesen auszuwerten. Für die Operatoren sind dann jeweils nur endliche Teilmengen, so genannte *Fenster*, sichtbar. Die Fenster werden über die Zeit hin um ein gewisses *Verschiebungsintervall* verschoben. Die Fenstergröße und das Verschiebungsintervall werden dem Operator bei Initialisierung mitgeteilt und müssen letztendlich vom Anfragesteller spezifiziert werden.

Die Fenstergröße bleibt bei dem oben beschriebenen Ansatz während der Anfrageauswertung konstant. [TMSF03] präsentiert einen Ansatz, mit dem der potentiell unendliche Datenstrom in endliche Teilstücke unterschiedlichster Größe unterteilt werden kann. Die Idee hierbei ist es, den Datenstrom mit Markierungsobjekten, so genannten *Interpunktionen (Punctuations)* zu versehen. Damit werden die Grenzen der Teilstücke explizit im Datenstrom definiert und nicht wie bei dem Fenster-Ansatz von der Parametrisierung der Operatoren festgelegt. Eine Interpunktion beinhaltet ein Prädikat, das für alle der Interpunktion nachfolgenden Tupel nicht mehr gilt. Temperatursensoren könnten zum Beispiel jede Stunde eine Interpunktion in den Datenstrom einfügen, um nachfolgende Operatoren darauf hinzuweisen, dass in Zukunft keine Tupel auftreten können mit einem Zeitstempel, der kleiner als der Zeitstempel der Interpunktion ist. Ein nachfolgender Join-Operator, der alle Tupel verknüpft, die eine maximale Zeitdifferenz von 1 Stunde haben, kann nun alle Tupel, die älter als eine Stunde sind, aus seinem lokalen Speicher löschen. An dieser Stelle wird klar, dass das Prädikat der Interpunktion und das Prädikat der im Join enthaltenen Selektion auf irgendeine Weise korreliert sein müssen. Ein Interpunktionsprädikat, das zusichert, dass kein Temperaturwert kleiner als z.B. 20°C übermittelt wird, wäre im obigen Fall nicht von Nutzen, denn die Join-Bedingung enthält die Zeitdifferenz und nicht die Temperatur. Weiterhin muss zugesichert sein, dass die eintreffenden Interpunktionen auch tatsächlich zu der geforderten

	einmalig	kontinuierlich
vordefiniert	(1): Zeichne am 24.12.2008 um 16 Uhr einen Kartenauschnitt mit dem Radius von 1km um den Hauptbahnhof Stuttgart	(2): Zeichne einen Kartenausschnitt mit dem Radius von 1km, um die aktuelle Position einer Person und aktualisiere die Karte jede Minute.
ad hoc	(3): Zeichne sofort einen Kartenauschnitt mit dem Radius von 1km um den Hauptbahnhof Stuttgart	(4): Aktualisiere den Radius von Anfrage (2) von 1km auf 2km.

Tabelle 2.1.: Beispielanfragen an ein strombasiertes Visualisierungssystem

Dezimierungsrate der zwischengespeicherten Tupel führt. Ein Interpunktionsprädikat, das nur zur Löschung von 10 Objekten im lokalen Speicher des Operators führt, während in der Zwischenzeit 100 neue Objekte hinzugekommen sind, würde nicht ausreichen. Dies im Einzelnen nachzuprüfen und gegebenenfalls spezielle Interpunktionsoperatoren, die die geforderten Interpunktionen generieren, an der richtigen Stelle im Operatorbaum einzufügen, wäre die Aufgabe des Anfrageprozessors (s.u.). Dies generell für den Anfrageprozessor vorauszusetzen, würde dessen Entwicklung erschweren. Deswegen wird in dieser Arbeit darauf verzichtet, diesen Ansatz umzusetzen. Jedoch die Idee, den Datenstrom mit Markierungsobjekten zu versehen, hat auch in dieser Arbeit Anwendung gefunden (siehe hierzu Kapitel 6.2).

2.3.4. Anfragearten

In der Literatur [BBD+02] werden zudem noch verschiedene Anfragearten differenziert. Es lassen sich *einmalige* und *kontinuierliche* Anfragen unterscheiden. Eine einmalige Anfrage entspricht dem Request-Response-Paradigma. Die Anfrage wird hierbei einmalig über den zum Zeitpunkt der Anfrageberechnung gültigen Daten ausgewertet und die Antwort an den Anfragesteller gesendet. Kontinuierliche Anfragen werden über die Zeit bearbeitet, Ergebnisse werden somit auch über die Zeit hin produziert und an den Anfragesteller zurückgesendet. Weiterhin lassen sich *vordefinierte* und *Ad-hoc*-Anfragen unterscheiden. Eine vordefinierte Anfrage ist dem Anfragebearbeitungssystem vor der Anfragebearbeitung bekannt. Das heißt, das System hat noch Zeit, gewisse Optimierungen durchzuführen, bevor Ressourcen endgültig an die Anfragebeantwortung gebunden werden. Ad-hoc-Anfragen sind dem System nicht vorher bekannt. Sie referenzieren Datenströme, die schon begonnen haben, und deren Elemente womöglich schon verworfen wurden. Vier Beispielanfragen für die eben beschriebenen Kategorien lassen sich in Tabelle 2.1 betrachten.

2.3.5. Data Stream Management Systeme (DSMS)

Systeme für die Beantwortung von Anfragen über kontinuierlichen Datenströmen werden *Data Stream Management Systems (DSMS)* genannt. Dieses kann prinzipiell aus mehreren Knoten, die die Datenströme verarbeiten, aufgebaut sein. In der Regel existiert eine zentrale Instanz, der *Anfrageprozessor*, an dem die Anfragen von Anfragestellern eintreffen. Der Anfrageprozessor muss dabei keine physische zentrale Instanz bilden, sondern kann natürlich auch auf eine beliebige Art und Weise verteilt realisiert werden. Im Folgenden wird bildlich von einem einzigen System ausgegangen, nämlich von dem Punkt, an dem Anfragen eintreffen. Der Anfrageprozessor muss aus den Anfragen den *logischen Operatorgraphen* berechnen. Dieser enthält die für die Verarbeitung benötigten Operatoren in einer speziellen Anordnung. Der Graph ist eine logische Sicht auf die zur Verarbeitung benötigten Komponenten. Er trifft keinerlei Aussagen darüber, auf welchen Knoten später die Berechnungen durchgeführt werden, oder welche physischen Implementierungen der Operatoren verwendet werden. Der logische Operatorgraph kann durch Mittel der *logischen Optimierung*, oder auch *algebraischen Optimierung*, optimiert werden. Dies beinhaltet in der Regel ein Umordnen und Zusammenfassen der Operatoren durch gewisse Transformationsregeln, die die Semantik der Anfrage nicht beeinflussen. Ist der entgültige logische Operatorgraph gefunden, muss aus den möglicherweise verschiedenen Implementierungen eines logischen Operators die passende ausgewählt werden und entschieden werden, wo diese ausgeführt werden. Dieser Teil der Optimierung wird *physische Optimierung* genannt. Das Resultat dieser Optimierung ist der *physische Operatorgraph*, der Informationen darüber enthält, auf welcher physischen Komponente welche konkrete Implementierung eines Operators ausgeführt werden soll. Da Anfragen in einem solchen System in der Regel kontinuierlich sind, kann und sollte dieser Prozess iterativ während der Anfragebeantwortung wiederholt werden. Denn die Lasten an den einzelnen Komponenten können sich über die Zeit ändern, was eine Umschichtung der Last (*Lastverteilung)* zwischen den physischen Entitäten des Systems nötig machen könnte. Bei der Beantwortung mehrerer Anfragen ist eventuell eine gemeinsame Nutzung der Ressourcen zwischen den Anfragen möglich, weshalb auch hier eine logische und physische Optimierung zumindest beim Hinzukommen oder Wegfallen einer Anfrage sinnvoll erscheint.

2.4. Nexus

Nach [BDG+04] ist NEXUS eine offene Plattform für kontextbezogene Anwendungen. Man nennt eine Anwendung *kontextbezogen*, wenn sie sich an den Kontext, in dem sie ausgeführt wird, anpasst. Der Kontext kann zum Beispiel der Ausführungsort der Anwendung sein, oder auch diverse Sensoren oder Geräte, welche für die Anwendung zum Ausführungszeitpunkt verfügbar sind. Damit Anwendungen unmittelbaren Zugriff auf ihren Kontext haben, bzw. auf einfache und einheitliche Weise diesen erkennen können, wurde ein globales *Kontextmodell (Augmented World Model)* entwickelt, das eine virtuelle Repräsentation der physischen Welt darstellt. Anwendungen können kontextbezogene Daten oder Dienste mit Hilfe dieses Kontextmodells nutzen. Zum Beispiel könnten einem Museumsbesucher

Informationen über ein Austellungsstück auf einem Handheld-Gerät, wie einem PDA oder Mobilfunktelefon, abhängig von seiner aktuellen Position angezeigt werden. Es sind mannigfaltige Daten oder Dienste in diesem Anwendungsspektrum denkbar. Da es nicht sinnvoll wäre, die in diesem Zusammenhang anfallenden, potentiell sehr großen Datenmengen in einem zentralen Datenspeicher unterzubringen, verfolgt NEXUS eine Strategie der verteilten Speicherung der Daten. Aufgabe der NEXUS-Plattform ist es, die verteilten Daten und Dienste in dem globalen Kontextmodell zu föderieren. Dies geschieht für die Anwendungen völlig transparent. Das heißt, für die Anwendung besteht eine konsistente Sicht auf die Daten des Kontextmodells, die sich aber je nach Ausführungskontext ändern kann. Dem Museumsbesucher zum Beispiel stehen die Informationen über die Ausstellungsstücke erst dann zur Verfügung, wenn dieser die Eingangspforte des Museums passiert hat. Befindet sich der Besucher noch außerhalb des Geländes, so sind die verfügbaren Informationen möglicherweise auf den Namen, die Anfahrtsbeschreibung und die Eintrittspreise des Museums beschränkt. Dabei bildet die Position des Besuchers möglicherweise nicht die einzige zu beachtende Kontextinformation. Die Frage, ob der Besucher schon eine Eintrittskarte erstanden hat, könnte weiterhin die Zugriffsmöglichkeit auf Informationen beeinflussen.

Der Kontext einer Anwendung muss nicht notwendiger Weise aus statischen Informationen wie Eintrittspreisen oder Anfahrtsbeschreibungen bestehen. In einer Welt der allgegenwärtigen Computer nach [Mat03] sind Anwendungen denkbar, die ihr Umfeld mit Hilfe von Sensoren untersuchen und gegebenenfalls auch verändern. Für viele Anwendungsfelder ist es nur bedingt sinnvoll, die großen, potentiell unbeschränkten Datenmengen, die bei der Überwachung der Umwelt durch Sensoren angehäuft werden, in einem großen Datenspeicher zu speichern. Die Menge und Art der Daten, die zwischengespeichert werden muss, hängt von der jeweiligen Anwendung ab. Wird zum Beispiel der Durchschnitt der letzten zehn Temperaturwerte eines Thermometers berechnet, so müssen auch nur diese Werte zwischengespeichert werden. Somit muss die NEXUS-Plattform eine strombasierte Verarbeitung von Daten unterstützen (siehe 2.3). [Sar08] schlägt hierfür eine Peer-to-Peer-Architektur vor, die so genannte *Datenstrom-Föderation*, deren wichtigsten Aspekte in Abschnitt 2.4.2 erläutert werden.

2.4.1. Augmented World Model (AWM)

Das *Kontextmodell (Augmented World Model)* nach [NM01] ist ein objektorientiertes Datenmodell. Objekte der realen Welt können mit Hilfe des Kontextmodells modelliert werden. Insbesondere die Positionen der Objekte werden berücksichtigt, da dies eine Kontextinformation für eine breite Klasse von kontextabhängigen Anwendungen ist. Neben realen Objekten kann das Modell zusätzlich mit virtuellen, in der realen Welt nicht vorhandenen Informationen angereichert werden.

Um die virtuellen und realen Informationen des Kontextmodells zu modellieren, wird eine auf XML basierende Sprache, die *Augmented World Model Language (AWML)*, verwendet. Diese wird auch verwendet, wenn Entitäten der Nexus-Plattform Informationen über Objekte des Kontextmodells austauschen. Die statischen Teile des Kontextmodells werden in so genannten *Spatial Model Server* (siehe Abschnitt 2.4.1) abgelegt. Die Anfragesprache

Augmented World Query Language (AWQL) wird verwendet, um Anfragen an einen Spatial Model Server zu stellen. AWQL ist eine Anfragesprache für einmalige Anfragen. Um kontinuierliche Anfragen zu unterstützen, müsste AWQL entsprechend erweitert werden, damit zum Beispiel die Definition von Fenstern ermöglicht wird.

Nexus-Datenmodell

Das UML-Diagramm des von NEXUS verwendeten Datenmodells ist in Abbildung 2.3 zu sehen. Ein *Nexus-Objekt* kann mehrere *Attribute* besitzen, wobei Attributinstanzen mit dem gleichen Namen mehrfach vorkommen können (man spricht hier auch von *Mehrfach-Attributinstanzen*). Ein Attribut kann entweder einen einfachen Wert besitzen oder aus mehreren *Attributteilen* bestehen. Zusätzlich kann ein Attribut mit Meta-Informationen in Form von mehreren Attributteilen versehen werden. Damit ist es ein objektorientiertes Modell mit der Möglichkeit, hierarchische Attributstrukturen zu bilden, welches allerdings auf eine Hierarchie von maximal zwei Attribut-Ebenen beschränkt ist.[4] Das objektorientierte Modell eignet sich, um die heterogenen Objekte der realen und virtuellen Welt abzubilden.

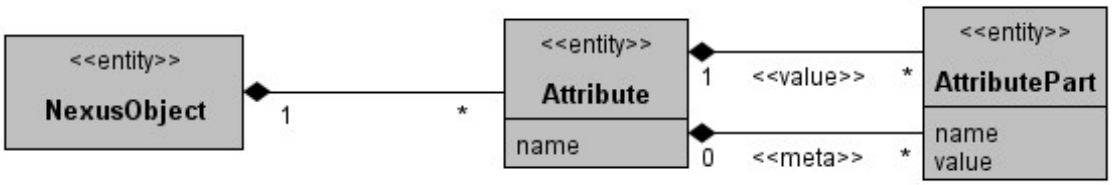


Abbildung 2.3.: Datenmodell von Nexus

Augmented World Schema (AWS)

Für die Spezifikation, welche konkrete Typen mit welchen Attributen existieren können und dürfen, wird das so genannte *Augmented World Schema (AWS)* verwendet. Das AWS wird mit Hilfe mehrerer XML-Schema-Dateien [W3C04] beschrieben. Eine davon ist das NEXUS *Standard Class Schema (NSCS)*. Hier wird die Struktur aller Basis-Objekttypen des Kontextmodells beschrieben, also prinzipiell welche Attribute innerhalb eines Objektes vorkommen dürfen und welche nicht. Anwendungen oder Dienste können diese Struktur erweitern, indem sie so genannte *Extended Class Schemas (ECS)* definieren. In Abbildung 2.4 ist ein Beispiel für die Erweiterung eines des Objekt-Typs *StaticObjekt* mit Hilfe eines ECS zu erkennen. In diesem Fall wird durch Vererbung der Objekt-Typ *Exhibit* (Ausstellungsstück) gebildet, der zusätzlich das Attribut `information` besitzt. Dies ist beispielsweise sinnvoll, wenn ein Museum das Kontextmodell mit Objekten, die die Austellungstücke repräsentieren, anreichern will. Die Erweiterungen zum NSCS sind aber anwendungsspezifisch und werden

[4] Die Beschränkung auf zwei Attribut-Ebenen hat keinen zwingenden technischen Grund, sondern ist lediglich der aktuelle Stand des Datenmodells, das zu Beginn des NEXUS-Projektes nur schlichte Attribute vorsah.

deswegen von anderen Anwendungen möglicherweise nicht verstanden. Das NSCS hingegen ist für alle NEXUS-Anwendungen obligatorisch. Damit sind für andere Anwendungen die Informationen des NSCS zugänglich. Im Beispiel sind das die Attribute `id`, `name` und `pos`, während nur für das Museum das Attribut `information` sichtbar ist.

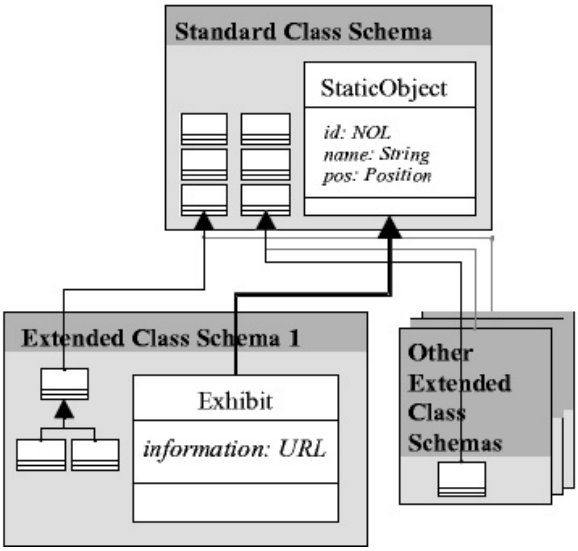


Abbildung 2.4.: Nexus Standard Class Schema [NM01]

Neben der Struktur der Typen des Kontextmodells werden mit Hilfe des AWS noch die Struktur der Attribute (ob einfach oder komplex) und die atomaren Basistypen der Attributwerte festgelegt. Dies geschieht mit Hilfe der XML-Schemas *Nexus Standard Attribute Schema (NSAS)* und *Nexus Standard Attribute Types (NSAT)*. Für beide dieser XML-Schemata besteht wiederum die Möglichkeit, sie durch entsprechende Extended Schemas zu erweitern.

Spatial Model Server (SpaSe)

Um statische Teile des Kontextmodells, wie Gebäude oder Straßen, zu speichern, werden so genannte *Spatial Model Server* verwendet. Daten des Kontextmodells, die mit einer hohen Dynamik verbunden sind, wie Sensordaten, sollten sinnvollerweise nicht in einem Spatial Model Server abgelegt werden. Jedoch können gewisse Meta-Informationen über Sensoren in einem Spatial Model Server hinterlegt werden. Diese Meta-Informationen könnten zum Beispiel die Position des Sensors sein, sowie Informationen, die der Anwendung helfen, Verbindung mit einer Sensor-Komponente aufzunehmen und so Datenwerte zu extrahieren. Jeder Spatial Model Server deckt ein gewisses geographisches Dienstgebiet ab, die so genannte *Augmented Area*. Die Dienstgebiete und die jeweils dazugehörigen SpaSes werden in einem für alle Komponenten zugänglichen Verzeichnis hinterlegt. Dies könnte entweder zentral oder nach [Sar08] als verteilter Dienst der Datenstrom-Föderation realisiert werden.

Nexus Object Locator (NOL)

Der NEXUS *Object Locator* ist ein global eindeutiger Identifizierer für die Objekte des Kontextmodells. Er besteht aus einer Endpunkt-Information des zugehörigen Spatial Model Servers, auf dem sich das Objekt befindet, einem Identifizierer der Augmented Area (*AAID*) und einem Objekt-Identifizierer (*OID*). Die OID identifiziert global eindeutig ein virtuelles oder reales Objekt des Kontextmodells. Falls sich Augmented Areas überlappen, können Objekte in mehreren Augmented Areas vorkommen und damit auch in mehreren Spatial Model Servern vorliegen und mit möglicherweise unterschiedlichen Informationen versehen werden. Dabei bleibt aber die OID des Objektes gleich, während sich die Endpunkt-Information und die AAID unterscheidet. So kann eine Anwendung leicht feststellen, ob unterschiedliche Repräsentationen des gleichen Objektes des Kontextmodell vorliegen, indem sie die OIDs vergleicht.

2.4.2. Datenstrom-Föderation

Die *Datenstrom-Föderation (Stream Federation)* wird durch eine 3-Schicht-Architektur realisiert. Wie in Abbildung 2.5 zu erkennen, finden sich die kontextabhängigen Anwendungen in der obersten Ebene wieder. Sie stellen kontextbezogene, möglicherweise kontinuierliche Anfragen an die *Nexus-Anwendungsschnittstelle (Nexus Application Interface)*. Dabei ist es möglicherweise sinnvoll, verschiedene anwendungs- und domänenspezifische Anfragesprachen zuzulassen. Die *Datenstrom-Föderationsebene (Stream Federation layer)* ist dafür zuständig, die für die Beantwortung der Anfragen benötigten Daten zu beschaffen. Die Entitäten der Datenstrom-Föderation, die so genannten *Knoten (Nodes)*, kontaktieren über die *Nexus-Dienstschnittstelle (Nexus Service Interface)* die Entitäten der Dienstebene, um die für die Beantwortung der Anfrage benötigten Daten zu beschaffen. Auch an dieser Stelle werden je nach realisiertem Dienst verschiedene Anfragesprachen benötigt.

Die Knoten bilden einen lose gekoppelten Peer-to-Peer-Verbund. In einem solchen Verbund sind die Knoten in der Regel auf beliebige Art und Weise untereinander verbunden. Im Gegensatz hierzu steht die so genannte Client-Server-Architektur wie beispielsweise innerhalb des WWW. Hierbei kommunizieren viele Clients mit einem zentralen Server. Die Kommunikation zwischen den Clients findet gegebenenfalls über den zentralen Server statt.

Die Abbildung 2.5 zeigt den Datenfluss einer kontinuierlichen Anfrage, welche sowohl statische Daten aus einem Kontextserver (Spatial Model Server), als auch dynamische Daten von Sensoren einbezieht. Durchgezogene Pfeile stehen dabei für einen einmaligen Datenfluss, während die gestrichelten Pfeile einen kontinuierlichen Datenstrom darstellen.

Abbildung 2.6 liefert eine detailliertere Ansicht der oben angedeuteten kontinuierlichen Anfrage. Zu Beginn stellt ein *Client* eine Anfrage an den *Föderations-Manager (Federation Manager)*. Der Föderations-Manager ist die zentrale Komponente des Systems und verantwortlich dafür, die Anfrage der konkreten Anfragesprache zu interpretieren und alle benötigten

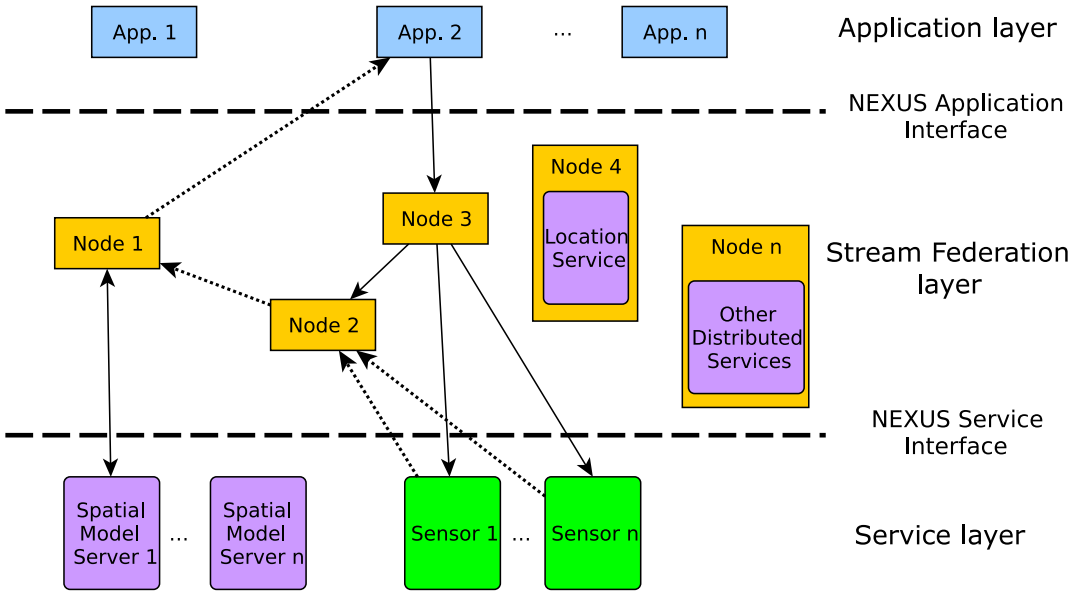


Abbildung 2.5.: Nexus Architektur [Sar08]

Schritte zu unternehmen, dass diese auch beantwortet wird. Hierzu ist eventuell eine logische und physische Optimierung sinnvoll und nötig (siehe 2.3.5). Wurde entschieden, auf welchem Knoten welcher Operator ausgeführt werden soll, werden gewisse *Setup-Dateien* an die Knoten gesendet. Diese werden von der *Kontroll-Schnittstelle (Controll Interface)* empfangen und anschließend werden entsprechend der Anweisungen in den Setup-Dateien die benötigten Operatoren innerhalb so genannter *Boxen* (siehe hierzu Abschnitt 2.4.2) auf dem jeweiligen Knoten installiert. In den Setup-Dateien sind auch die Verknüpfungen zwischen den Boxen und zu den *Quellen (Sources)* und *Senken (Sinks)* definiert. Eine Quelle bezeichnet innerhalb eines Knotens eine Entität, die Daten aussendet, während eine Senke Daten empfängt. In der Abbildung sind zudem noch spezielle Quellen und Senken angegeben: die *Plattform-Quelle (Platformsource)* und die *Plattform-Senke (Platformsink)*. Erstere empfängt Daten von anderen Knoten, während die zweitere Daten an andere Knoten versendet. Aus der Abbildung ist nicht ersichtlich, auf welche Art parallel mehrere Anfragen von mehreren Clients verarbeitet werden können. Hierfür muss es möglich sein, mehrere Operatoren unabhängig voneinander auf einem Knoten auszuführen. Für die virtualisierte Ausführung der Operatoren auf den Knoten wurden schon Konzepte und Techniken entwickelt, die hier jedoch nicht detailliert beschrieben werden. Informationen hierzu sind in [Sar08] zu finden. Für diese Arbeit genügt die Annahme, dass Operatoren mehrerer Anfragen voneinander abgeschottet auf einem Knoten ausgeführt werden können.

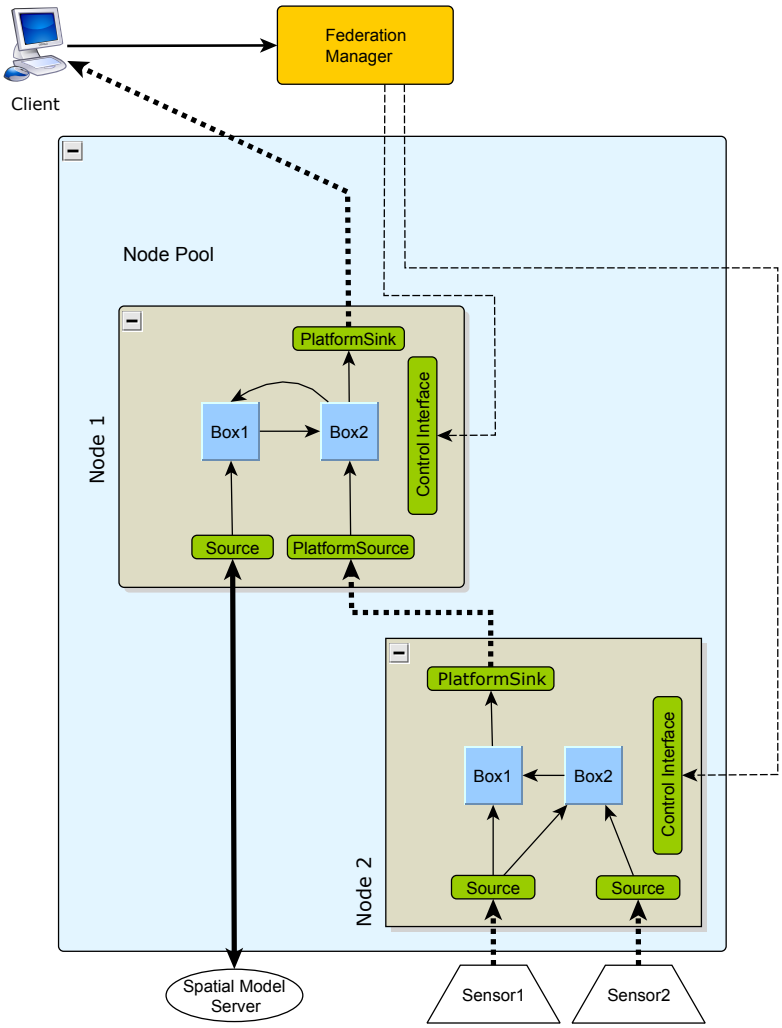


Abbildung 2.6.: Die NEXUS Datenstrom-Föderation (ähnl. [Sar08]).

Operator-Modell

Operatoren werden innerhalb so genannter *Boxen* in einem Knoten ausgeführt. Nach [Sar08] ist eine Box eine vollständige, unabhängige und aktive Ausführungseinheit. Abbildung 2.7 zeigt die Bestandteile einer Box. Eine Box enthält einen *Operator*, einen *Eingangsverwalter (Input Manager)* und so viele *blockierende Warteschlangen (Blocking Queues)* wie Eingänge (Inputs).

[Sar08] verwendet durchgängig den Begriff „Operation“. Dies spiegelt sich natürlich auch in den Benennungen der Klassen und den hier zitierten Abbildungen wieder (zum Beispiel in Abbildung 2.7). Der Begriff „Operation“ wird in [Sar08] gleichbedeutend mit dem hier verwendeten Begriff „Operator“ verwendet. Hier wird der Begriff „Operator“ verwendet, wenn ganz allgemein die Entität mit der Verarbeitungslogik gemeint ist. „Operation“ wird

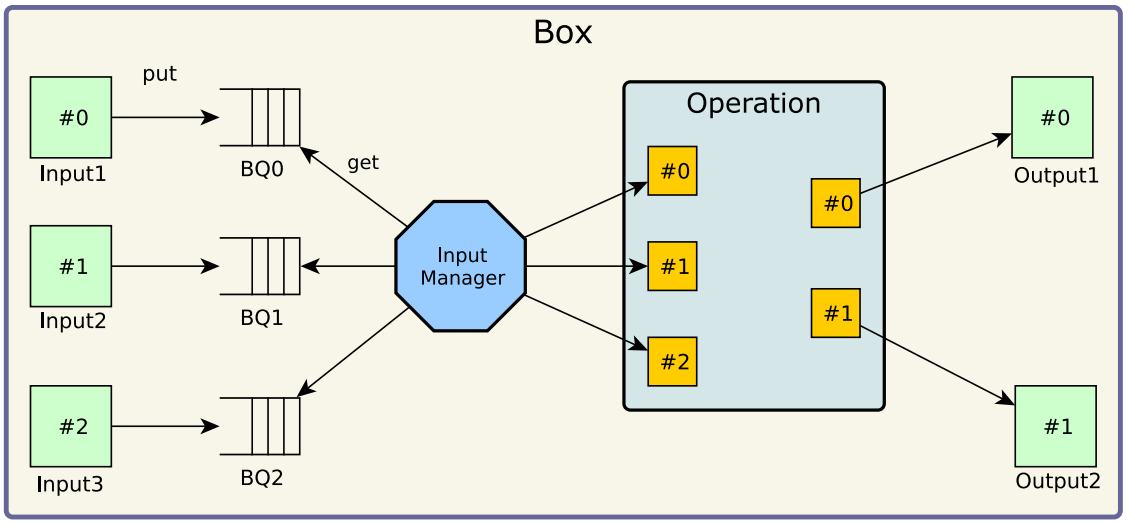


Abbildung 2.7.: Aufbau einer Box [Sar08]

nur dann verwendet, wenn die konkrete Java-Schnittstelle `Operation` gemeint ist, wie in Abbildung 2.8 erkennbar. Ein *Operator* ist nach [Sar08] eine multi-funktionale, passive Datentransformationseinheit mit mehreren Ein- und Ausgängen. Ein *Operator* muss die Java-Schnittstelle `Operation` implementieren, damit die Plattform den Operator ausführen kann.

In Abbildung 2.8 wird die Vererbungshierarchie einer beispielhaften Implementierung der `Operation`-Schnittstelle `SomeOperation` aufgezeigt.[5] Wie zu sehen, gibt es vier zentrale Methoden. Die Methode `setProperty(String, Object)` macht es möglich, die Operatoren zu Beginn oder während der Laufzeit mit Parametern zu versehen, bzw. diese zu ändern. Das Setzen der Parameter kann während der Initialisierung stattfinden, bevor die `init`-Methode aufgerufen wird. Dies geschieht genau dann, wenn die Kontroll-Schnittstelle des Knotens eine Setup-Datei vom Federation Manager erhält. Durch das Senden von Update-Dateien können Parameter auch während der Laufzeit geändert werden. Dann wird die Methode `setProperty` zu einem beliebigen Zeitpunkt aufgerufen. Möchte eine Implementierung auf das Setzen von Parametern reagieren, so muss sie die `setProperty`-Methode überschreiben. Die Methode receive(int, Object) wird vom InputManager aufgerufen, wenn dieser entscheidet, dass eine `Operation`-Instanz ein Objekt von einem bestimmten Eingang (`inputID`) bearbeiten soll. Die Standard-Implementierung der `AbstractOperation` delegiert den Aufruf direkt an die `process`-Methode, welche die eigentliche Verarbeitungslogik enthält. Gelegentlich ist es sinnvoll, die Daten zwischen den Eingängen einer Operation zu synchronisieren. Ein Beispiel hierfür wäre ein Additions-Operator, der die Summe der Objekte zweier Eingänge bildet. In diesem Fall sollte ein Operator die Methode `receive` überschreiben und die Synchronisation durchführen. Operatoren können entweder direkt die

[5] Der Übersicht wegen wurden einige Methoden in der Abbildung weggelassen. Eine vollständige Darstellung ist in [Sar08] zu finden.

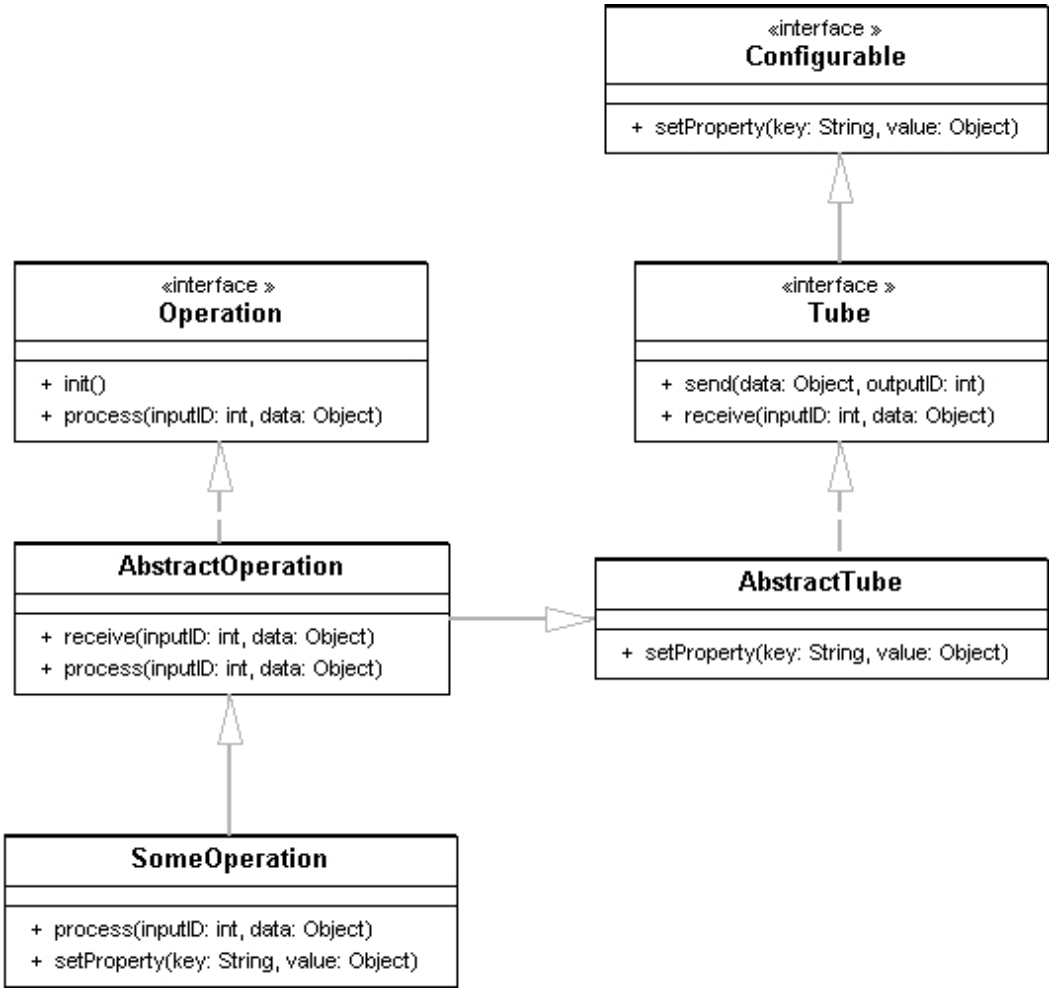


Abbildung 2.8.: Klassendiagramm einer Realisierung der `Operation`-Schnittstelle

`Operator`-Schnittstelle realisieren oder die abstrakte Klasse `AbstractOperation` verwenden. Diese liefert Standard-Implementierungen für viele der zu implementierenden Methoden. Letztlich können `Operation`-Implementierungen erstellt werden, indem lediglich die Klasse `AbstractOperation` erweitert und die Methode `process(input:int, data:Object)` implementiert wird.

Eine *Blocking Queue* ist eine Warteschlange mit einer bestimmten maximalen Kapazität, die spezielle Methoden für das Hinzufügen und Entfernen von Elementen hat (vgl. u. a. [SM08b] oder [Ull07]). Es stehen Methoden mit verschiedenen Laufzeitverhalten zur Verfügung. So gibt es Hinzufüge-Methoden, die blockieren, wenn die Warteschlange voll ist, und Methoden, die sofort zurückkehren, ohne das Element hinzugefügt zu haben. Für das Entfernen von Elementen existieren analog blockierende Methoden und nicht-blockierende Methoden, die entweder warten, bis wieder ein Element Warteschlange ist, oder sofort zurückkehren. Elemente werden am Kopf der Schlange entfernt und am Ende der Schlange hinzugefügt. Die Elemente stehen damit in einer *First-In-First-Out-Ordnung (FIFO-Ordnung)*. Die konkrete

Implementierung der Warteschlange kann in der Setup-Datei ausgewählt werden. So ist es auch möglich, Warteschlangen mit anderem Laufzeitverhalten auszuwählen. Beispielsweise könnte es nützlich sein, Elemente mit einer bestimmten Strategie zu verwerfen, wenn die Warteschlange schon ihre maximale Kapazität erreicht hat und weitere Elemente ankommen, anstatt den Sender zu blockieren. Weiterhin könnte hier schon eine gewisse Sortierung der ankommenden Elemente vorgenommen werden.

Der *Input Manager* ist die einzige aktive Komponente innerhalb einer Box. Er ist dafür zuständig, die *Operation* mit Arbeit zu versorgen. Hierfür bildet der Input Manager in der Regel einen Ausführungsstrang (*Thread*) pro Eingang und entfernt ein Element aus der entsprechenden Warteschlange und gibt es der Operation zur Verarbeitung. In der Regel ist es sinnvoll, dass der Ausführungsstrang eines Einganges blockiert, solange die Operation ein Element der Eingangswarteschlange verarbeitet. Es existieren jedoch auch andere sinnvolle Verhaltensweisen für den Input Manager, weshalb [Sar08] auch die Möglichkeit vorsieht, innerhalb einer Setup-Datei eine bestimmte Implementierung eines Input Managers auszuwählen.

2.5. Forschungsstand

TerraVision [SJR+04] ist ein hardwareunterstützter Ansatz, der es ermöglicht, hochauflösende Videoströme über Gigabit-Netzwerke zu senden. Die Daten werden in unkomprimierter Form übermittelt, was jedoch spezielle Hardware-Anforderungen nach sich zieht. Denn das Versenden von Daten mit einer Bitrate von mehreren Gigabits pro Sekunde würde die Rechenleistung der Sendermaschine nahezu ausschöpfen. Deswegen sieht [SJR+04] spezielle *TerraVision Boxen* vor, welche einen VGA- oder DVI-Eingang besitzen und spezielle Hardware, die den Videostrom des VGA- oder DVI-Signals über ein Gigabit-Netzwerk übertragen kann. Damit kann die Sendermaschine seinen VGA- oder DVI- Ausgang mit der Box verbinden und somit hochauflösende Videoströme produzieren, ohne die CPU zu behelligen. Auf der Empfängerseite befindet sich wiederum ein Gerät, welches den Videostrom aus dem Gigabit-Netzwerk in ein VGA- oder DVI-Signal umwandelt. Neben dem Senden und Empfangen von Videoströmen werden hier auch Synchronisationsfragen zwischen den Videoströmen gelöst. So ist es, durch die Verwendung mehrerer Boxen unter Anderem auch möglich, so genannte *Tiled-Displays* zu realisieren. Ein *Tiled-Display (gekachelter Bildschirm)* ist ein Verbund von Anzeigegeräten, der aus einzelnen Anzeigegeräten besteht. Durch eine synchronisierte, verteilte Bildgenerierung entsteht die Illusion eines einzigen hochauflösenden Anzeigegerätes.

Für die Visualisierung von großen Datenmengen schlägt [SB03] die Visualisierung mit Hilfe einer Grid-Infrastruktur vor. Durch die Verteilung des Visualisierungsprozesses auf eine Grid-Infrastruktur soll es möglich werden, komplexere Szenarien zu visualisieren. Hierfür müssen unabhängige Komponenten entwickelt werden, die gewissen Standards entsprechen, um von einem speziellen Rahmenwerk ausgeführt zu werden. Es wird prognostiziert, dass der Erfolg eines solchen Rahmenwerks von gewissen Grundvoraussetzungen abhängt. Dies ist zum Beispiel die Integration heterogenener Komponenten, aus denen der Benutzer

die für seine Anwendung beste auswählen kann bzw. die Erweiterungsmöglichkeit mit benutzerdefinierten Komponenten.

Der *GVK (Grid Visualization Kernel)* [KRHP04] ist ein realisierter Ansatz eines solchen Rahmenwerks zur verteilten Visualisierung basierend auf einer Standard-Middleware für Grids. Die Komponenten der Visualisierungspipeline werden von einem Grid ausgeführt. Dabei wurden zwei Aufteilungsmöglichkeiten einer verteilten Visualisierungspipeline realisiert, nämlich die Erzeugung eines Videostroms (siehe Abbildung 2.2a) und die Erzeugung von Geometriedaten (siehe Abbildung 2.2b) mit Hilfe des Grids.

[ADK+99] beschreibt eine Plattform namens *Dv (Distributed Visualization)* zur verteilten Ausführung einer Visualisierungspipeline auf einer Grid-Infrastruktur. Hier wird die Idee von so genannten *aktiven Frames* eingeführt. Ein aktiver Frame beinhaltet Programmtext und Daten, welche von Rechner zu Rechner weitergeleitet werden und dort ausgeführt werden. Die Idee hinter diesem Ansatz ist, dass die Bilddaten eines Bildes des resultierenden Videostroms in einen solchen Frame gepackt und sukzessive verarbeitet werden. Problematisch an diesem Ansatz ist das fortwährende Weiterleiten von Programmtext. Insbesondere für Aufgaben, welche die Bildgenerierung betreffen, werden in der Regel umfangreiche Bibliotheken benötigt, welche dann entweder für jeden Frame erneut übertragen werden müssten oder der Grid-Infrastruktur auf eine andere Art zur Verfügung gestellt werden. Zudem besitzen Daten in verschiedenen Frames in der Regel hohe Redundanzen. Das Frame-basierte Verarbeitungsmodell erschwert es, Redundanzen zwischen den Frames zu entfernen.

Grundsätzlich besteht die Frage, welche Schritte der Visualisierungspipeline verteilt und welche lokal auf dem Client durchgeführt werden. Wird die Bildgenerierung lokal durchgeführt, so muss sichergestellt werden, dass der Client hierzu die notwendige Leistungsfähigkeit mit sich bringt. Wird die Bildgenerierung von einer anderen Maschine durchgeführt, so sind die Interaktionsmöglichkeiten eingeschränkt. Bei beiden Möglichkeiten muss die Gesamtlast der Bildgenerierung von einer Komponente bewältigt werden. [GSJ+06] geht einen Kompromiss ein und verteilt die Gesamtlast der Bildgenerierung innerhalb einer Client-Server-Architektur. Nach clientseitiger Spezifikation eines Kegelstumpfes (*Frustum*) beginnt der Server die 3D-Szene in ein Tiefen- und Farbild zu rendern. Die Bilder werden dann an den Client gesendet, der aus dem Tiefen- und Farbild ein dreidimensionales Punktbild erzeugt. Daraus wird ein zweidimensionales, anzeigbares Bild erstellt, indem mit Hilfe der *Splatting*-Technik Farbpunkte im 3D-Raum im übertragenen Sinne auf die Bildfläche geworfen werden und dort zerplatzen. Damit müssen weniger Informationen übertragen werden, da die dreidimensionalen Punkte vom Client zwischengespeichert werden. Außerdem hängt die Bildrate dann von der lokalen Splatting-Geschwindigkeit des Clients ab und nicht von der Bildgenerierungsgeschwindigkeit des Servers.

Der zuerst genannte Ansatz TerraVision beschäftigt sich mit der verteilten Bildgenerierung und mit Synchronisationsproblemen, die hierdurch entstehen. Die verteilte Bildgenerierung ist kein zentraler Aspekt dieser Arbeit. Der zuletzt genannte Ansatz beschreibt eine interessante Technik, mit deren Hilfe die Menge an den zu übertragenden Daten in einem verteilten Umfeld reduziert werden kann. Der Ansatz basiert allerdings auf einer Client-Server-Architektur, die in dieser Arbeit nicht verfolgt wird. Die übrigen Ansätze ähneln dem

Ansatz der hier verfolgt wird, nämlich der verteilten, virtualisierten Ausführung einer Visualisierungspipeline. Als Verteilungs-Infrastruktur wird dort allerdings ein Grid gewählt. Ein *Grid* ist nach [Fos02] *„ein System, das Ressourcen koordiniert, die keiner zentralen Instanz untergeordnet sind, und das standardisierte, offene und universale Protokolle und Schnittstellen benutzt, um nicht triviale Dienstgüte bereitzustellen"*. Dieser Definition ist kein schneidender Unterschied zu einem P2P-System, wie es für diese Arbeit verwendet wird, abzugewinnen. Der Übergang ist eher fließend, was auch daran liegt, dass Grid-Systeme immer mehr Methoden und Algorithmen vieler P2P-Systeme übernehmen, um deren Skalierbarkeit zu verbessern. Ein differenzierendes Kriterium könnte vielleicht die unterschiedliche Zielsetzung der Systeme sein. Während Grid-Systeme meistens darauf bedacht sind, eine bestimmte Dienstgüte, wie eine bestimmte Antwortzeit, oder einen bestimmten Durchsatz zu garantieren. Hierbei wird häufig auch spezialisierte Hardware, zum Beispiel im Bereich der Hochgeschwindigkeits-Netzwerke, verwendet. P2P-Systeme hingegen vertreten eher den Ansatz, eine möglichst flexible und skalierbare Struktur unter gleichberechtigten Entitäten herzustellen, um damit gewisse Ressourcen, wie zum Beispiel Dateien, auszutauschen. Dabei liegt der Fokus eher auf Skalierbarkeit und Flexibilität als auf die Bereitstellung einer gewissen Dienstgüte und in Folge dessen wird auch keine spezialisierte Hardware vorausgesetzt, um an einem P2P-Verbund teilzunehmen. Das heißt jedoch nicht, dass es prinzipiell unmöglich wäre, eine gewisse Dienstgüte bereitzustellen, oder dass in einem P2P-Netz keine spezialisierte Hardware vorkommen kann. Vielmehr können in P2P-Netzen heterogene Hardware-Systeme zusammenarbeiten und genau diese Tatsache kann auch ausgenützt werden, um gewisse Aufgaben effizienter zu lösen, oder eine gewisse Dienstgüte zur Verfügung zu stellen. Die flexible Struktur eines P2P-Verbundes ist für das kontextbezogene Anwendungsspektrum eher geeignet als eine Grid-Strukur. Hier kommt es vor Allem darauf an, dass Anwendungen sich schnell und einfach an einen neuen Kontext anpassen können, und dass unterschiedlichste Geräte miteinander kommunizieren können.

Kapitel 3

Entwurf

Nachdem die Architektur und das Komponentenmodell der Ausführungsumgebung schon in [Sar08] festgelegt wurden, bleibt nur noch zu entscheiden, welche Funktionalität für eine verteilte Visualisierungspipeline benötigt wird und welche Komponenten mit welcher Funktionalität versehen werden. Weiterhin zu entscheiden ist, wie diese Komponenten untereinander agieren und kommunizieren. Dabei ist zu beachten, welche Anforderungen an ein Visualisierungssystem bestehen könnten und die Aufteilung der Funktionalität so zu realisieren, dass möglichst viele der Anforderungen umgesetzt werden können.

3.1. Anforderungen

Ziel der Visualisierung ist es, dem Benutzer einen Videostrom zu präsentieren und zusätzlich Interaktionsmöglichkeiten zu bieten. Ein Videostrom wird als flüssig wahrgenommen, wenn die Bilder in einer Rate von in etwa 25 Bildern pro Sekunde angezeigt werden. Liegt die Bildwiederholfrequenz darunter, so werden die bewegten Bilder als ruckartig empfunden. Fällt die Frequenz unter ein bestimmtes Minimum, so können keine Bewegungen mehr wahrgenommen werden. Das Bild wird als sprunghaft empfunden. Ab dann ist eine sinnvolle, bewegte Visualisierung nicht mehr möglich. Wann dies passiert, hängt von der Schnelligkeit der Bewegungen im Bild ab. Damit hängt es von dem aktuellen Visualisierungsszenario, bzw. von den Ansprüchen des Benutzers ab, mit welcher Rate das Video wiedergegeben werden soll. Die Bildwiederholfrequenz kann also nicht als „harte“ Anforderungen festgelegt werden, wohl aber die Möglichkeit, Einfluss auf den Visualisierungsprozess zu nehmen. Wird die Visualisierung zum Beispiel als zu ruckartig empfunden, sollte die Möglichkeit bestehen, die Auflösung zu verkleinern, um so eine flüssigere Darstellung zu erhalten.

Eine Einflussnahme auf den Visualisierungsprozess soll sowohl bei der initialen Anfrage an das Visualisierungssystem stattfinden können, als auch während der Laufzeit einer Visualisierung. Ein Benutzer könnte beispielsweise eine Anfrage stellen, die die Visualisierung einer Stadt in einem Radius von 1km um das Stadtzentrum zur Folge hat. Während die Visualisierung im Gange ist, könnte er durch Navigieren innerhalb des visualisierten Stadtszenarios den Ausschnitt der Visualisierung sukzessive verschieben.

Für die oben genannte Navigation entstehen spezielle Anforderungen an die Reaktionszeit des Visualisierungssystems. Nach [Nie93] liegt die maximal erlaubte Reaktionszeit eines Systems bei $0,1$ Sekunden, damit der Benutzer die Reaktion des Systems als sofortig empfindet. Eine Einhaltung dieser Grenze ist dann sinnvoll, wenn der Benutzer für die Navigation ein Eingabegerät besitzt, welches den Eindruck vermittelt, kontinuierliche Stimuli an das System senden zu können. Ein Beispiel hierfür wäre das kontinuierliche Drücken einer Cursor-Taste für eine Bewegung des Bildes in eine bestimmte Richtung, oder die Navigation mit Hilfe einer Maus. Liegt die Reaktionszeit darüber, so ist keine sinnvolle Navigation mit solchen Eingabegeräten möglich. Der Benutzer würde durch konstantes Drücken zu viele Stimuli auslösen und so in vielen Fällen über das beabsichtigte Ziel hinausschießen. In diesem Fall wäre es sinnvoller, die Navigation über Buttons in einer graphischen Oberfläche zu realisieren. Diese könnten dann blockieren und als gedrückt erscheinen, solange eine Navigationsanfrage durchgeführt wird. In [Nie93] ist hierfür eine maximale Obergrenze von einer Sekunde angeben. Wird diese überschritten, sollte der Benutzer darauf hingewiesen werden, dass das System gerade dabei ist, den Wunsch des Benutzers umzusetzen.

3.2. Entwurfsdiskussion

In Abbildung 3.1 ist die logische Sicht einer beispielhaften Umsetzung der verteilten Visualisierungspipeline zu sehen. Zu Beginn schickt der Betrachter (Client) eine Visualisierungsanfrage an den Föderations-Manager (Federation Manager). Diese Anfrage wird mit Hilfe einer Anfragesprache spezifiziert. Die Spezifikation dieser Anfragesprache ist nicht Teil dieser Arbeit. Wie diese Anfragesprache beschaffen sein könnte und ob es eventuell sinnvoll ist, für jedes Anwendungsfeld eine eigene Anfragesprache zuzulassen, wird in 3.2.8 diskutiert. Aus der Anfrage wird auf irgend eine Weise der logische Operatorgraph der Anfrage extrahiert. Der Federation Manager schickt für die Verwaltung der Operatoren Setup-, Update-, und Delete-Befehle an die entsprechenden Knoten, welche die Operatoren ausführen. Die verschiedenen Schritte der Visualisierungspipeline können mit Ausnahme des Bildgenerierungs-Schrittes von mehreren Operatoren durchgeführt werden. Das Ergebnis des Selektions-Schrittes (*Selection*) und des Anreicherungs-Schrittes (*Enhancement*) sind *Nexus-Objekte (NexusObjects)*. Ergebnis der Geometrieerzeugung sind *visuelle Objekte (VisualObjects, siehe 3.2.7)*. Sie sind eine spezielle Art der Nexus-Objekte und repräsentieren die geometrischen Formen, die während der Geometrieerzeugung erzeugt wurden. Das Ergebnis Rendering-Komponente ist ein binärer Videostrom, der an den Client gesendet wird und dort angezeigt wird (Dies entspricht der Aufteilungsmöglichkeit nach Abbildung 2.2a). Die Bilderzeugung ist ein komplexes Aufgabenfeld, für das oftmals spezielle Anforderungen an Hardware und Software gelten. Das gleiche gilt für das Erstellen, Versenden und Abspielen von Videoströmen. Deswegen soll für diese beiden Aufgaben auf schon vorhandene Technologie zurückgegriffen werden (siehe Kapitel 4).

Die Reihenfolge der Selektion, Anreicherung, Geometrieerzeugung und Bildgenerierung ist dabei nicht zwingend festgelegt. Zwar erscheint die genannte Reihenfolge intuitiv als vernünftig, jedoch lassen sich auch Beispiele finden, bei denen es angemessen ist, von dieser Reihenfolge abzuweichen. So könnte zum Beispiel nach der Geometrieerzeugung ein

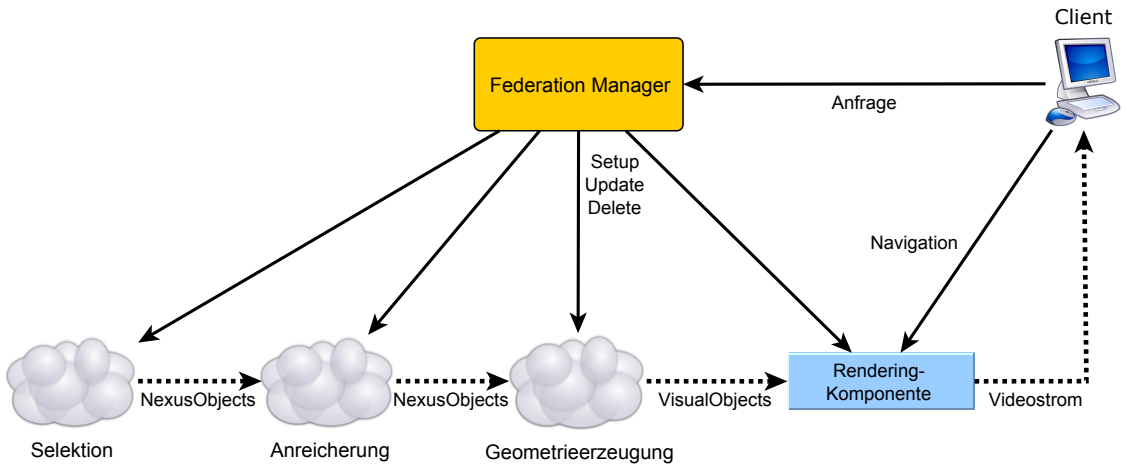


Abbildung 3.1.: Beispielhafte Umsetzung einer verteilten Visualisierungspipeline

Datenstrom, welcher rote und grüne Objekte enthält, nach der Farbe selektiert werden, so dass einem Client ein Videostrom präsentiert wird, der nur rote Objekte enthält, während ein anderer Client nur die grünen Objekte zu Gesicht bekommt. Durch das Mehrfachverwenden von Teilen des Operatorbaums für mehrere Clients können kostbare Ressourcen gespart werden.

3.2.1. Interaktion

Um der in Abschnitt 3.1 beschriebenen Anforderung der Interaktionsmöglichkeit Genüge zu tun, muss es die Möglichkeit geben, Visualisierungsanfragen zu aktualisieren. Anfragen, die eine schon laufende kontinuierliche visuelle Anfrage aktualisieren, können prinzipiell mit der gleichen Syntax gestellt werden wie die zu aktualisierende Anfrage. Es muss lediglich ein Mechanismus entwickelt werden, der es erlaubt, eine zuvor gestellte Anfrage zu referenzieren. Ein Beispiel hierfür wäre ein Identifizierer, der vom Föderations-Manager zugewiesen wird und dem Client beim Stellen der Anfrage als Antwort gesendet wird. Möchte nun ein Client eine schon laufende, früher gestellte Anfrage aktualisieren, so muss er seine Änderungswünsche in einer Anfragesprache formulieren und eine spezielle *Update-Anfrage* an den Föderations-Manager senden, welche den Identifizierer der ursprünglichen Anfrage enthält, sowie die in der Anfragesprache formulierten Änderungswünsche. Dieser hat nun lokal die ursprüngliche Anfrage gespeichert und kann durch Vergleich mit den Änderungswünschen die benötigten Parameteränderungen an den entsprechenden Operatoren durch Senden von Update-Befehlen an die Knoten durchführen.

Die gerade beschriebene Vorgehensweise hat jedoch einen gravierenden Nachteil. Die Aufgaben, die der Föderations-Manager in diesem Fall durchführen muss, sind relativ komplex. Zum Einen müssen die Änderungswünsche mit der schon vorhandenen Anfrage verglichen werden. Weiterhin könnte es sein, dass die Änderungswünsche eines Clients die Ressourcen

bzw. Möglichkeiten eines bestimmten Knotens überschreiten. Dadurch ist eine Migration des Operators von einem Knoten auf einen anderen Knoten notwendig. Auch wenn dies nach [Sar08] weitestgehend transparent für den Client durchgeführt werden kann, so macht sich zumindest die hierfür benötigte Zeit für den Client bemerkbar. Nach [Sar08] benötigt das Installieren eines simplen Operators auf einem Knoten ca. 0,2 Sekunden. Bedenkt man nun, dass möglicherweise mehrere Operatoren verschoben werden müssen, scheint eine Reaktionszeit, die im Sekundenbereich liegt, nicht unwahrscheinlich. Dies ist zu langsam, um dem Benutzer eine mühelose Navigation zu ermöglichen. Wenn erst nach Sekunden feststeht, wohin die Reise nach der Durchführung einer Navigationsanfrage ging, macht dies das Navigieren äußerst schwierig. Deswegen erscheint es sinnvoll, eine direkte Kontrollverbindung zur Rendering-Komponente einzurichten. In Abbildung 3.1 ist dies durch den direkten Pfeil vom Client zur Rendering-Komponente ausgedrückt. Damit ist es Aufgabe des Clients zu entscheiden, in welchen Fällen eine Update-Anfrage an den Föderations-Manager geschickt wird und wann die direkte Kontrollverbindung zum Rendering-Komponente genutzt wird. Navigiert sich der Betrachter zum Beispiel an den Rand der Datenmenge, welche im Selektions-Schritt ausgewählt wurde, so wäre es wünschenswert, wenn dieses die Client-Software automatisch erkennt und die Selektion der Daten entsprechend der aktuellen Position im Bild mit Hilfe einer Update-Anfrage aktualisiert.

Der oben beschriebene Ansatz verbessert zwar die Reaktionszeit, jedoch kann immer noch nicht das Gefühl von unmittelbarer Interaktivität vermittelt werden. Dieses wird benötigt, um die Navigation mit Hilfe von Eingabegeräten zuzulassen, welche eine kontinuierliche Steuerung zulassen (siehe 3.1). Das Kernproblem ist hier die Tatsache, dass der Knoten, der die Bildgenerierung durchführt, sehr weit von dem eigentlichen Betrachter entfernt sein können und dadurch Latenzen entstehen können. Befindet sich die Rendering-Komponente auf der anderen Seite des Globus und unter der Annahme von einer Signalgeschwindigkeit nahe der Lichtgeschwindigkeit, entstehen hier schon Latenzen von mehr als $0,1$ Sekunden. Hinzu kommt noch die Tatsache, dass das unterliegende Netzwerk, wie das Internet, in der Regel ein so genanntes *Best-Effort-Netzwerk* ist. Diese Klasse von Netzwerken geben keinerlei Garantien hinsichtlich der Latenzen (vgl. hierzu auch [Tan03]). Dies hat zum Einen zur Folge, dass zusätzlich zur reinen Signallaufzeit eine unbekannte Zeit kommt, die das Netzwerk benötigt, um die Datenpakete an die richtige Stelle zu vermitteln. Zum Anderen entsteht beim Abspielen des vom Betrachter empfangenen Videostroms der Bedarf, empfangende Datenpakete eine gewisse Zeit zwischenzuspeichern bevor sie abgespielt werden, um so die Varianz der Latenzen der Datenpakte, den so genannten *Jitter*, auszugleichen. Um den Eindruck der unmittelbaren Interaktivität zu vermitteln, müssen Reaktionszeiten von weniger als $0,1$ Sekunden erreicht werden und dies in einem beliebigen verteilten Umfeld, kaum zu realisieren. Die unmittelbare Folge hiervon ist, dass es zum Erreichen von unmittelbarer Interaktivität notwendig wird, die Bildgenerierung vom Client durchführen zu lassen, falls hohe Netzwerklatenzen auftreten können. Abbildung 3.2 zeigt eine solche Aufteilung. Das Visualisierungsnetz erzeugt dann lediglich die Geometriedaten (siehe hierzu auch Abbildung 2.2b).

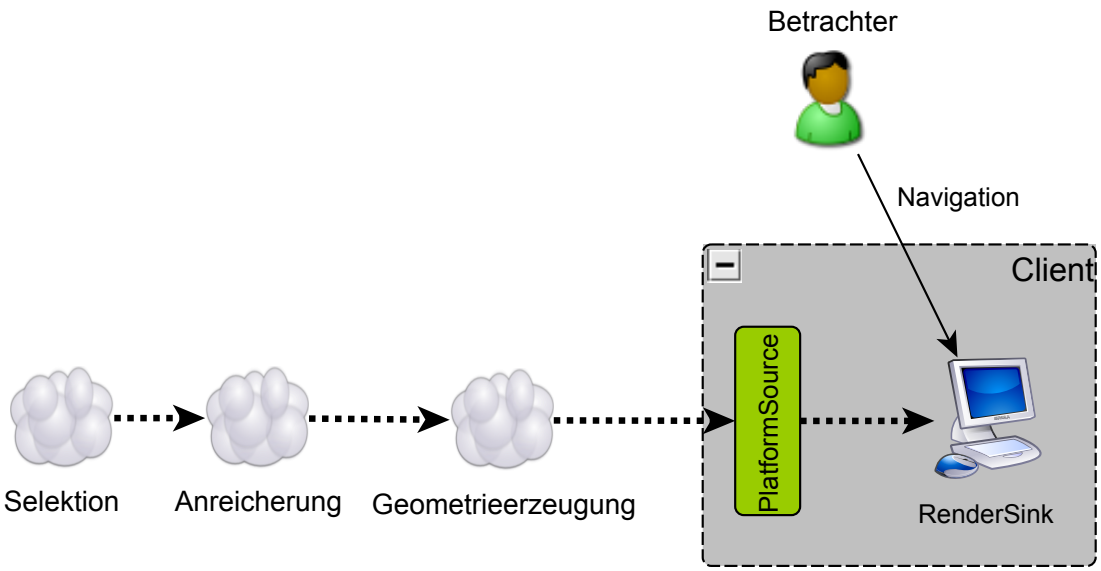


Abbildung 3.2.: Clientseitige Umsetzung der Bildgenerierung

3.2.2. Datenmodell und Datenformat der ausgetauschten Objekte

Wie weiter oben schon angedeutet, werden die Rohdaten, die aufbereiteten Daten und die Geometriedaten in dem von NEXUS verwendeten Datenformat ausgetauscht (siehe Kapitel 2.4.1). Die Verwendung ein- und desselben Datenmodells hat verschiedene Vorteile, aber auch einen Nachteil, was im Folgenden erläutert wird.

Für den Entwickler von Operatoren hat die Verwendung des gleichen Datenformats den Vorteil, dass die Komponenten, welche für das Serialisieren und Deserialisieren der Daten zuständig sind, nur einmal entwickelt werden müssen. Würde bei jedem Schritt des Visualisierungsprozesses ein anderes Datenformat entstehen, so müsste der Entwickler für jedes dieser Formate Serialisierer und Deserialisierer bereitstellen. Zusätzlich steht hinter jedem Datenformat ein bestimmtes Datenmodell, für das Bibliotheken geschrieben werden müssen, um auf die Daten zuzugreifen, oder sie manipulieren zu können. Diese Bibliotheken müssten dann mit den Operatoren mitgeliefert werden. Außerdem erleichtert die Verwendung eines einheitlichen Datenmodells die Entwicklung der Operatoren. Denn der Entwickler muss sich nur in die Bibliothek eines Datenmodells einarbeiten.

Daneben lässt ich die Abbildung der Daten auf die geometrischen Formen leichter formulieren, wenn ein einheitliches Datenmodell verwendet wird. Entsprechend Abbildung 3.3 kann die Abbildung eines Daten-Objektes auf eine geometrische Form als Funktion angesehen werden, die eine eine beliebige Menge von Attributen auf eine beliebige Menge von Attributen abbildet.

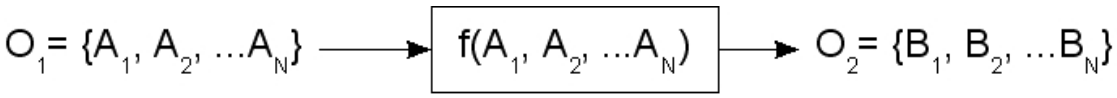


Abbildung 3.3.: Geometrieerzeugung als Funktion

Neben der Übertragung von einzelnen Objekten sollte es die Möglichkeit geben, Mengen von Objekten zu übertragen. Damit können Operatoren einbezogen werden, die wie die in der Datenbank-Welt gängigen relationalen Operatoren auf Mengen operieren. Für das Übertragen von Mengen, sowie das Übertragen von einzelnen Objekten, kann AWML verwendet werden. Dabei enthält die AWML-Datei bei der Übertragung eines einzelnen Objektes eben nur ein Objekt. Als Folge lässt sich vom Übertragungsformat kein Unterschied feststellen zwischen der Übertragung eines einzelnen Objektes und einer Objektmenge, welche nur ein einziges Objekte enthält. Es ist einzig aus dem Anfragekontext ersichtlich, ob ein Objekt oder eine Menge übertragen wurde.

Generische Datenmodelle, wie das Datenmodell von NEXUS, können für Daten unterschiedlichster Anwendungsfelder verwendet werden. Gelegentlich sind aber aus Effizienzgründen spezielle, auf die Anwendungsfelder zugeschnittene Datenmodelle und -formate sinnvoll. Das Übertragungsformat von NEXUS-Daten ist XML-basiert und XML hat die Eigenschaft, für wenig Information viel Platz zu benötigen. Bei einem Visualisierungprozess müssen aber häufig immense Datenmengen übertragen werden und XML ist hierfür denkbar ungeeignet. Eine Entwicklung eines neuen effizienteren Übertragungsformats für die NEXUS-Daten unter Beibehaltung des generischen Datenmodells wäre ein möglicher Ansatz zur Verbesserung der Effizienz (siehe hierzu Kapitel 7.2).

3.2.3. Koordinatensysteme der Bildgenerierung

Die virtuelle Welt, in der die visuellen Objekte angezeigt werden, wird im Folgenden *virtuelles Universum* genannt. Im virtuellen Universum ist ein kartesisches Koordinatensystem definiert. Die Koordinaten dieses Koordinatensystems werden *Weltkoordinaten* genannt. Jedes Objekt definiert ein eigenständiges Koordinatensystem, dessen Ursprung durch ein Positionsattribut festgelegt wird. Standardmäßig ist dieses gleich ausgerichtet wie das Weltkoordinatensystem, kann aber um beliebige Achsen rotiert werden. Während das Positionsattribut in Weltkoordinaten vorliegt, sind alle anderen Koordinaten innerhalb eines Objektes, wie zum Beispiel der Eckpunkt eines Würfels, als relativ zum Positionsattribut aufzufassen und werden im Folgenden *Objektkoordinaten* genannt. Blickwinkel und die Position des Betrachters wird in einem weiteren Koordinatensystem, dem so genannten *Sichtkoordinatensystem*, modelliert. Dieses Koordinatensystem hat seinen Ursprung in der Mitte des Bildschirms, die X-Achse wird vom Betrachter aus nach rechts abgetragen, die Y-Richtung nach oben und die Z-Achse senkrecht in Richtung des Betrachters. Damit blickt der Betrachter in Richtung der negativen Z-Achse. Die X- und Y-Achse des Sichtkoordinatensystems definieren die Projektionsebene der Bildfläche, die Z-Achse gibt den Abstand des Betrachters von der Projektionsebene an. Letzten Endes gibt es noch das zweidimensionale Bildkoor-

dinatensystem der Bildfläche, das identisch ist mit der durch das Sichtkoordinatensystem definierten Projektionsebene.

Für eine Darstellung auf der zweidimensionalen Ebene der Bildfläche müssen die verschiedenen Koordinatenssysteme sukzessive ineinander überführt werden. Die dafür benötigten Schritte lassen sich in einer bestimmten Folge, der so genannten *Rendering*-Pipeline (siehe u. a. [Zep04], anordnen. Die Objektkoordinaten müssen in Weltkoordinaten überführt werden, die Weltkoordinaten in Sichtkoordinaten und die Sichtkoordinaten in Bildkoordinaten. Für die notwendigen Transformationen und die Anzeige der transformierten Objekte wird eine Grafikbibliothek benötigt, die in Kapitel 4 ausgewählt wird.

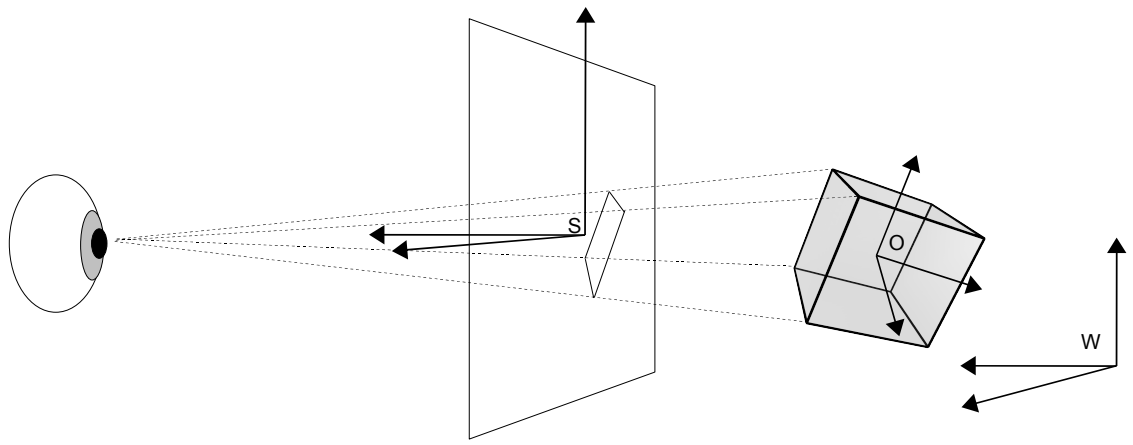


Abbildung 3.4.: Koordinatensysteme der Bilderzeugung. „W“ steht für Weltkoordinaten-, „O“ für Objektkoordinaten- und „S“ für Sichtkoordinatensystem.

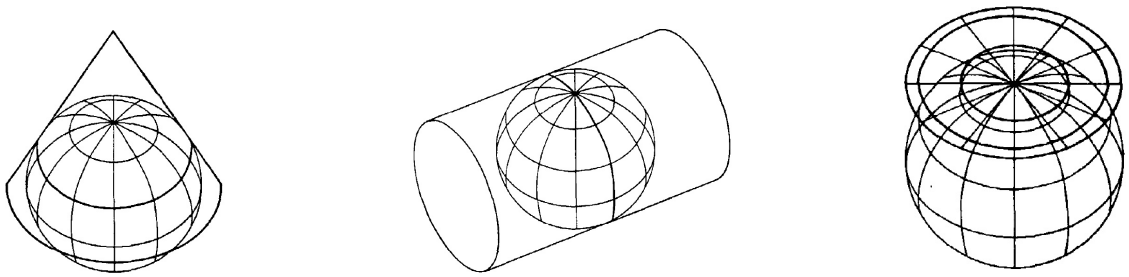
Alle oben beschriebenen Koordinatensysteme sind kartesisch. Für manche Anwendungen werden andere Koordinatensysteme, zum Beispiel geographische wie WGS-84, benötigt. Im Folgenden werden einige geographische Transformationsverfahren vorgestellt und erörtert, ob diese Art von Transformationen von der Rendering-Komponente durchgeführt werden sollte.

3.2.4. Geographische Koordinatentransformation

Die im NEXUS-Umfeld auftretenden Daten sind sehr häufig mit einer Position verknüpft. Diese Position wird üblicherweise in einem globalen Koordinatensystem wie WGS-84 angegeben. Solche Koordinatensystem basieren oft auf einem ellipsoiden Modell der Erde. Koordinaten, die in einem solchen Koordinatensystem angeben sind, eignen sich nicht direkt für eine Visualisierung. Deshalb ist für die Visualisierung von Daten, die in einem globalen Koordinatensystem referenziert sind, eine Transformation notwendig. Transformationen bzw. Projektionsverfahren existieren in zahlreicher Form (vrgl. hierzu auch [Sny87]). Im Folgenden sollen sollen die drei am häufigsten auftretenden Klassen erläutert werden und

welche Implikationen die Präsenz verschiedener Verfahren für die Visualisierungspipeline hat.

Bei allen hier dargestellten Projektionsverfahren wird vom Erdinneren auf eine Projektionsebene projiziert. Dabei wird bei der *Kegel-* und *Zylinderprojektion* die Projektionsebene in Form eines Kegels, beziehungsweise eines Zylinders, um den Erdkörper gelegt. Bei der *azimutalen Projektion* wird die Projektionsebene tangential an den Erdkörper angelegt.



(a) Kegelprojektion

(b) Zylinderprojektion

(c) Azimutale Projektion

Abbildung 3.5.: Transformationsverfahren [Sny87]

Von den oben beschriebenen Projektionsverfahren kann von vorne herein kein optimales ausgewählt werden. Denn bei allen Projektionsverfahren tritt eine gewisse Verzerrung bei der Projektion in die Ebene auf. Betrachtet man zum Beispiel Abbildung 3.5b, so fällt auf, dass eine Verzerrung in auf der Ost-, West-Achse entsteht, während Längenangaben auf der Nord-, Süd-Achse längentreu bleiben. Würde man nun die Projektionsachse so um den Erdkörper legen, dass die Projektionsebene am Äquator den Erdkörper berühren würde, würde eine Verzerrung in Nord-, Süd-Richtung auftreten. Die letztere Methode der Zylinderprojektion wird *Mercator Transformation* genannt, während die zuerst genannte Methode als die *transversale Mercator Transformation* bezeichnet wird. An der Vielfalt der existierenden Verfahren kann man erkennen, dass das Projektionsverfahren der Wahl von dem aktuellen Anwendungsfall abhängt. Das Projektionsverfahren, dass die Visualisierungspipeline verwendet, hängt also davon ab, was auf welche Weise visualisiert werden soll. Dieses wiederum ist inhärent anfrageabhängig. Es gibt sogar noch weitere Faktoren, die die Wahl des Projektionsverfahrens beeinflussen könnten. Betrachten wir zum Beispiel ein mobiles Objekt, welches sich anfänglich in Richtung Norden bewegt, dann aber ostwärts. Für die erste Bewegungsrichtung wäre also eine Projektion nach der transversalen Mercator Methode optimal, während man für die zweite die reguläre Mercator Methode verwenden würde. Sollen also mobile Objekte visualisiert werden, so ist es möglicherweise nötig, in verschiedenen Koordinatensystem zu arbeiten und während der Laufzeit einer Visualisierungsanfrage Transformationen in verschiedene Koordinatensysteme vorzunehmen.

Die Koordinatentransformation könnte natürlich von der Rendering-Komponente durchgeführt werden. Dies kostet jedoch auch Zeit, die während der Bildgenerierung nicht unbedingt zur Verfügung steht. Um die Rendering-Komponente nicht zusätzlich mit der Koordinatentransformation zu beschäftigen, wird angestrebt, die Transformation der Koordinaten in

einen externen Operator durchzuführen. Werden mehrere Anfragen von vielen Clients gleichzeitig verarbeitet, so müssen möglicherweise viele Transformationen in unterschiedliche Koordinatensysteme durchgeführt werden. Das heißt, es werden Objekte mit Koordinaten unterschiedlicher Koordinatensysteme ausgetauscht. Damit nun die verschiedenen Komponenten des Systems mit diesen unterschiedlichen Koordinaten umgehen können, so müssen die verschiedenen Transformationen klassifiziert und für alle Komponenten des Systems zugänglich gemacht werden.

[SHGN04] stellt einen Ansatz vor, der es ermöglicht, mit Koordinaten in verschiedenen Koordinatensystemen umzugehen. Für jede Transformation existiert eine eindeutige Kennziffer. Die Koordinaten werden nun, bevor sie ausgetauscht werden, mit dieser Kennziffer versehen. Beispiel 3.1 zeigt mit der Kennziffer „`4326`" versehenen Koordinaten im GML-Format [GML01]. Damit haben alle Komponenten die Möglichkeit, anhand der Kennziffer auf die verwendete Transformationsmethode rückzuschließen, und gegebenenfalls Transformationen von einem System in das andere vornehmen.

```
<gml:Point srsName="4326">
       <gml:coordinates> 10 20 </gml:coordinates>
</gml:Point>
```

Beispiel 3.1: Eine mit Kennziffer versehene Koordinate [SHGN04]

Natürlich müssten die Parameter der verschiedenen Transformationen in einem für alle Komponenten zugänglichen Verzeichnis eingetragen werden. Die Komponenten könnten dann anhand der Kennziffer feststellen, welches Transformationsverfahren mit welchen Parametern durchgeführt wurde. Die Schaffung eines solchen Verzeichnisses ist nicht Teil dieser Arbeit. Jedoch ist das oben beschriebene Austauschformat für Koordinaten in dieser Arbeit vorgesehen, um zukünftige Weiterentwicklungen zu ermöglichen. Einen konkreten Nutzen für diese Arbeit gibt es nicht. Um das dafür vorgesehene XML-Attribut `srsName` mit einem Attrappen-Wert zu versehen, wurde für diese Arbeit die neue Kennziffer „`40000`“ definiert. Dies bezeichnet ein lokales Koordinatensystem, für das keine Transformationseigenschaften zur Verfügung stehen. Es wird angenommen, dass zukünftig keine Transformationen mit dieser Kennziffer definiert werden.

3.2.5. Zustand der Operatoren

Ein strombasierter Operator beginnt bei der Ankunft eines neuen Objektes an einem Eingang mit dessen Verarbeitung, oder nimmt das angekommene Objekt in seinen lokalen Speicher auf. Dabei kann der Operator auch Ausgaben produzieren, das heißt, Objekte an seine Ausgänge schicken. Es könnte auch nützlich sein, den lokalen Speicher oder einzelne Objekte im lokalen Speicher des Operators zu verändern. Beispielsweise liest der Bildgenerierungs-Operator die visuellen Objekte ein und zeigt diese an. Jedoch sind die Objekte nicht nur zu sehen, wenn gerade ein visuelles Objekt verarbeitet wird, sondern kontinuierlich. Damit hält der Operator alle visuellen Objekte, die er gerade anzeigt, in seinem lokalen Speicher. Gelegentlich ist es nötig, diesen Speicher von außen zu ändern. Ein Beispiel hierfür wäre eine Positionsaktualisierung eines visuellen Objektes, oder das Löschen eines Objektes aus dem

Speicher, falls es nicht mehr angezeigt werden soll. Prinzipiell kann man den lokalen Speicher und seine aktuelle Arbeitsweise als den *Zustand* des Operators ansehen. Will man den Zustand eines Operators beeinflussen, so muss dies dem Operator mitgeteilt werden. Die Möglichkeit mit Operatoren zu kommunizieren, sieht die Datenstrom-Föderation durch das Setzen bzw. Aktualisieren von Parametern zur Laufzeit vor. Die Parameteränderungen werden jedoch in eigens dafür vorgesehenen Kontrollverbindungen an die Operatoren gesendet. Dies geschieht asynchron zur Verarbeitung der Datenströme. Dass heißt, es ist unmöglich vorauszusagen, an welchem Punkt eines Datenstrom die Arbeitsweise oder der Speicher eines Operators geändert werden. Will man eine Verhaltens- oder Zustandsänderung des Operators an einer bestimmten Stelle im Datenstrom, so kann nicht der Kommunikationsweg über Parameteränderungen gewählt werden. Außerdem können Parameter nur vom Föderations-Manager geändert werden und dies bedeutet zusätzlichen Overhead.

Deswegen ist es sinnvoll, eine zweite Möglichkeit vorzusehen, um mit den Operatoren zu kommunizieren. Durch das Senden von speziellen *Command*-Objekten kann auf den Speicher und die Arbeitsweise eines Operators während der Laufzeit eingewirkt werden. Die Kommando-Objekte können an einer bestimmten Stelle innerhalb eines Datenstroms eingefügt werden. Damit besteht eine explizite Verbindung zwischen dem Datenstrom und dem Ausführungszeitpunkt der *Command*-Objekte. Dies ähnelt dem Interpunktions-Konzept nach [TMSF03] (siehe Kapitel 2.3.3).

Die *Command*-Objekte lassen sich genau wie die Objekte der Modellierungssprache in einer Objekt-Hierarchie beschreiben, die mit Hilfe des *VisualClassSchema* definiert wurden. Diese wird im Folgenden inhaltlich beschrieben. Für die formale Spezifikation sei auf das *VisualClassSchema* im Anhang verwiesen (siehe Anhang A).

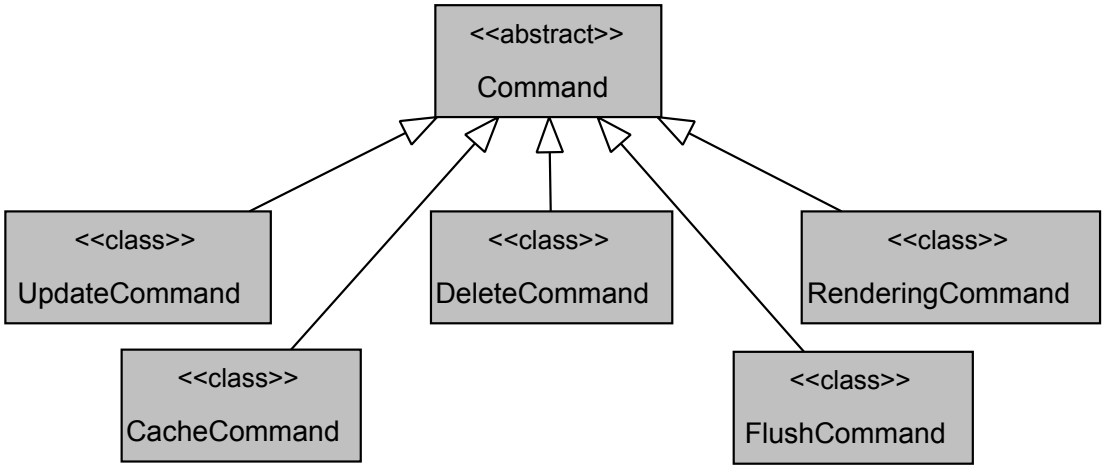


Abbildung 3.6.: Klassenhierarchie der Command-Klasse

Command

Alle Kommandos sind Unterklassen der *Command*-Klasse. Dadurch können Operatoren feststellen, ob es sich bei dem aktuellen Objekt um ein Kommando handelt oder nicht. Diese Klasse enthält keinerlei Attribute und dient lediglich der Markierung.

UpdateCommand

Ein *UpdateCommand*-Objekt zeigt einem Operator an, dass ein oder mehrere Attribute eines Objektes in dem lokalen Speicher aktualisiert werden sollen. Um das Objekt zu selektieren, enthält das *UpdateCommand* das `restrictionNol`-Attribut, dessen Wert die NOL des zu aktualisierenden Objektes spezifiziert. Weiterhin enthält das *UpdateCommand* ein oder mehrere `update`-Attribute. In diesen Attributen sind Informationen enthalten, welche Attribute mit welchen Werten aktualisiert werden sollen.

Abstrahiert man noch weiter, so können wir den Zustand eines Operators, der UpdateCommand-Objekte verarbeiten kann, als eine bestimmte Menge von *Nexus-Object*-Objekten charakterisieren. Eine direkte Folge hiervon ist, dass für eine Veränderung dieser Menge eine Anfragesprache wie AWQL verwendet werden kann. Die Schnittstelle, die ein Operator dann implementieren würde, wäre in diesem Fall die gleiche wie die eines Spatial Model Servers.

DeleteCommand

Ein *DeleteCommand*-Objekt löscht ein bestimmtes Objekt aus dem lokalen Speicher eines Operators. Um das Objekt zu selektieren, wird auch hier das `restrictionNol`-Attribut verwendet, welches das zu löschende Objekte auswählt.[1]

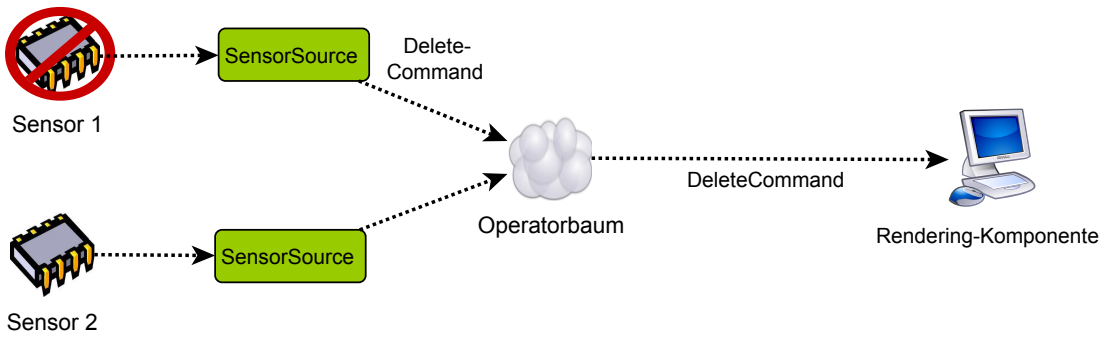


Abbildung 3.7.: Versenden eines DeleteCommands

[1] Auch hier könnte zur Selektion eine Anfragesprache wie AWQL verwendet werden.

In Abbildung 3.7 wird ein Anwendungsfall für das *DeleteCommand* aufgezeigt. Verschiedene Sensor-Quellen (*SensorSource*s) sind dafür zuständig, die Sensorinformationen aus Sensoren zu extrahieren. Die Sensorinformationen fließen durch einen beliebigen Operatorbaum an dessen Ende die Rendering-Komponente für die Visualisierung der Sensorinformationen sorgt. Nun versagt ein Sensor den Dienst, zum Beispiel weil dessen Batterie leer ist. Nur die angeschlossene *SensorSource* hat das Wissen darüber und kann nachfolgende Komponenten mit dem Senden eines *DeleteCommands* darüber informieren. Das *DeleteCommand* wird durch den gesamten an die Quelle angeschlossenen Teiloperatorbaum propagiert, so dass jede Komponente die Werte von dieser Quelle aus ihrem lokalen Speicher löschen kann. Letztendlich führt es dazu, dass die visualisierte Sensorinformation von der Bildfläche der Rendering-Komponente verschwindet.

RenderingCommand

Mit Hilfe des *RenderingCommand*s können Parameter des Rendering-Prozesses geändert werden. Das *RenderingCommand* wird nur von dem `RenderOperator` unterstützt. Momentan kann der Blick des Betrachters durch Translation oder Rotation geändert werden. Es ist jedoch durchaus denkbar, dass im Zuge von Weiterentwicklungen weitere Parameter hinzukommen.

CacheCommand

Manche Operatoren puffern Objekt für eine spätere Verarbeitung. Insbesondere Operatoren, die in irgendeiner Form mehrere Objekte aggregieren, verfahren nach diesem Schema. Es wird eine bestimmte Anzahl von Objekte gepuffert, dann anschließend aus den gepufferten Objekten ein aggregiertes Objekt erstellt. Das CacheCommand stellt die Möglichkeit dar einem Operator mitzuteilen, dass er mit der Pufferung der Objekte beginnen soll. Das *CacheCommand* besitzt keinerlei Attribute und dient lediglich der Markierung.[2]

FlushCommand

Mit dem *FlushCommand* kann die Verarbeitung gepufferter Objekte angestoßen werden. Das *FlushCommand* ist das Gegenstück des *CacheCommand*s (s. o.). Mit Hilfe dieser beiden Kommandos kann eine Aggregation von Objekten realisiert werden (siehe Kapitel 6.2). Auch die *FlushCommand*-Klasse dient lediglich der Markierung und enthält keinerlei Attribute.

[2] Das *Cache*- und *FlushCommand* wird in Kapitel 6.2 verwendet, um die Aggregation einer unabhängigen Punktmenge in ein Dreiecksnetz anzustoßen.

3.2.6. Typisierung von Datenströmen

In dem Umfeld der Datenstrom-Föderation können viele Arten von Datenströmen auftreten. Bei dem Selektionsschritt der virtualisierten Visualisierungspipeline entstehen Datenströme einer Art, die man typischerweise dem DSMS-Bereich zuordnet. Die Objekte dieser Datenströme, entsprechen einem gewissen Datenmodell. Das heißt, deren innere Struktur ist sowohl dem Operator-Entwickler als auch der Ausführungsumgebung bekannt. Es liegt auf der Hand, für diese Art von Datenströmen das Nexus-Datenmodell zu übernehmen. Die Objekte der Datenströme sind damit Instanzen des `GenericObjects`[3] und werden im Folgenden *Nexus-Ströme* genannt.

Wie in Kapitel 2.4.2 beschrieben, kann das Operator-Modell jedoch auch verwendet werden, um beliebige `Object`-Instanzen strombasiert zu verarbeiten. Damit könnten Binärdaten, die sich in abgeschlossene Einheiten unterteilen lassen, wie zum Beispiel Bilder, verarbeitet werden. Die innere Struktur dieser Objekte ist der Ausführungsumgebung nicht bekannt. Datenströme dieser Art werden im Folgenden *Objekt-Ströme* genannt.

Es gibt jedoch noch ein weiteres Verarbeitungsparadigma. Denken wir an den Videostrom einer Videokamera, der mit einem gängigen Codec, wie zum Beispiel MPEG-4 [ISO04a], kodiert wurde und über ein Netzwerk gesendet wird. In der Regel ist dem Operator-Entwickler die interne Struktur des MPEG-4-Stroms unbekannt. Das heißt, der Videostrom lässt sich aus Anwendungssicht nicht in logische Einheiten unterteilen.[4] Während die beiden oben genannten Datenstromarten einem nachrichtenbasierten Verarbeitungsparadigma entsprechen, ist hier die Abstraktion eines *transparenten Bitstroms* angebracht. Ein transparenter Bitstrom ist eine potentiell unendliche Folge von Bits, deren innere Struktur unbekannt ist. Datenströme dieser Art werden im Folgenden *Bitströme* genannt.

Es ist sinnvoll, die drei beschriebenen Datenstromarten auf unterschiedliche Granularität mit Typ-Information zu versehen. Während Bitströme nur auf Strom-Ebene mit Typ-Information versehen werden können, macht es eventuell Sinn, die Objekte der Objekt- und Nexus-Ströme auch mit Typ-Information zu versehen. Die Herausforderungen und der potentielle Nutzen, die bei einer Typisierung auf verschiedener Ebene entstehen, sollen in den folgenden Abschnitten erläutert werden.

Typisierung auf Strom-Ebene

Wenn ein Bitstrom mit einer Typangabe zu versehen wird, so könnte die Ausführungsumgebung automatisch eine Kompatibilitätsprüfung vornehmen. Beispielsweise besitzt ein bestimmter Client nicht die Mittel (software- oder hardwareseitig) einen Bitstrom zu

[3] Ein `GenericObject` ist eine Java-Klasse, mit deren Hilfe das generische Datenmodell von NEXUS auf Objekt-Ebene realisiert wird.

[4] Letzten Endes wird natürlich auch ein MPEG-4-Strom bei der Übertragung über ein paketvermitteltes IP-Netzwerk in Einheiten, nämlich Pakete, unterteilt. Dies ist jedoch Aufgabe des Codecs und geschieht transparent für den Benutzer der Codec-Komponente.

verarbeiten, der mit Hilfe eines bestimmten Codecs kodiert wurde. Die Ausführungsumgebung könnte nun automatisch einen Konvertierungsknoten zwischen der Bitstrom-Quelle und dem Client setzen, der eine entsprechende Konvertierung vornimmt. Hierfür müssten gewisse Regeln für die Operatoren definiert werden, die es der Ausführungsumgebung ermöglichen, an bestimmten Stellen automatisch Operatoren dazwischen zu schalten, ohne dabei die ursprüngliche Semantik der Anfrage zu beeinflussen. Das heißt, hierfür wäre eine semantische Klassifizierung der Operatoren notwendig, welche sich von Anwendungsfeld zu Anwendungsfeld unterscheiden kann. Bisher ungeklärt blieb die Frage, wo die Typinformation eines Bitstroms untergebracht wird. Die Typinformation in den Bitstrom selbst zu integrieren stellt sich als schwierig heraus, da die innere Struktur des Stroms unbekannt ist. Deswegen bietet sich hier an, Bitströme implizit zu typisieren. Das typische Vorgehen hierbei wäre es, die Ein- und Ausgänge der Operatoren mit einer Typinformation zu versehen. Dies könnte prinzipiell schon zur Entwicklungszeit mit Hilfe von Deskriptor-Dateien stattfinden.

Weiterhin kann es auch sinnvoll sein, Objekt- oder Nexus-Ströme mit einer Typangabe auf Strom-Ebene zu versehen. Dies würde bedeuten, dass alle Objekte des Stroms dem Typ des Datenstroms entsprechen. Ein Objekt-Strom vom Typ `SomeObject` würde dann zum Beispiel nur Instanzen der Java-Klasse `SomeObject` enthalten. Handelt es sich um einen Nexus-Strom, so sollte zusätzlich zu dem Java-Typsystem das Typisierungskonzept des AWS verwendet werden. Enthält ein Strom zum Beispiel nur Objekte vom Typ `GenericObject`, die den AWS-Typ *MobileObject* besitzen, so wäre zum Beispiel eine Typangabe der Form „`java:GenericObject|aws:MobileObject`“ angebracht.

Eine Typisierung auf Strom-Ebene ist genau dann nicht mehr ausreichend, wenn Datenströme Objekte mit verschiedenen Typen enthalten. Ein Datenstrom der Objekte vom AWS-Typ *MobileObject* und *Building* enthält, könnte zum Beispiel nur mit der gemeinsamen Oberklasse *NexusObject* typisiert werden. Solche hybriden Datenströme sind zum Beispiel das Resultat des in der DSMS-Welt gängigen *Union*-Operators, der eingehende Datenströme unabhängig ihres Typs vereinigt. Ansätze für eine feingranularere Typisierung werden im Folgenden beschrieben.

Statische Typisierung auf Objekt-Ebene

Eine Typisierung auf Objekt-Ebene bedeutet, dass jedes Tupel eine Typangabe enthält. Im Falle beliebiger Objekt-Ströme bietet die Java-Laufzeitumgebung ein Typsystem an, das hierfür benutzt werden kann. Die Objekte eines Nexus-Stroms enthalten zusätzlich noch die AWS-Typangabe. Möchten Operatoren diese Typinformation nutzen, so müssen sie hierfür die Augmented-World-Bibliothek verwenden. Der AWS-Typ lässt Rückschlüsse auf die Attributstruktur innerhalb des Objektes zu, was eine Validierung der Objekte des Strom möglich macht. Außerdem lässt es eine schnellere Verarbeitung verschiedener Objekttypen zu. Damit kann schon nach dem Lesen der Typinformation entschieden werden, was mit dem Objekt des entsprechenden Typs angestellt werden soll. Es kann unter Anderem auch eine Standard-Weiterleitung realisiert werden, bei der ein Operator Objekte bestimmten

Typs weiterleitet, ohne sie zu verarbeiten. Dies wird zum Beispiel dafür benötigt, gewisse Kommando-Objekte durch den Operatorbaum zu propagieren (siehe Kapitel 3.2.5).

Java-Typen werden zur Entwicklungszeit festgelegt. Sie sind in Form von Klassennamen fest im Java-Code definiert. Auch AWS-Typen können mit Hilfe der AWS-Schema-Dateien zur Entwicklungszeit festgelegt werden. Das heißt, die Typen sind zur Laufzeit statisch, weshalb hier auch von der *statischen Typisierung* gesprochen wird.

Die statische Typisierung ist dann nicht mehr angemessen, wenn die Operatoren die innere Struktur der Objekte ändern. Für Objekt-Ströme aus Java-Objekten ist dies ohnehin nicht möglich. Jedoch Objekte von Nexus-Strömen könnten solchen Strukturänderungen unterzogen werden. Man denke an dieser Stelle beispielsweise an die Funktionalität eines typischen DSMS. Mitunter können beliebige Attribute der Objekte durch Projektions- oder Verbund-Operatoren hinzukommen oder wegfallen. Die Struktur der Objekte wird erst zur Laufzeit festgelegt, ist also abhängig von der aktuellen Anfrage. Die Frage, ob in einem solchen Umfeld überhaupt noch eine Typisierung möglich und sinnvoll ist, wird im nächsten Abschnitt erörtert.

Dynamische Typisierung auf Objekt-Ebene

Die feingranularste Methode Datenströme zu typisieren, ist die dynamische Typisierung auf Objekt-Ebene. Operatoren, die dynamisch Typen erstellen, welche zur Entwicklungszeit noch nicht bekannt sind, könnten diese Typen zum Beispiel in ein zentrales Typ-Verzeichnis eintragen. Andere Komponenten könnten sich mit Hilfe dieses Verzeichnisses die dynamische Typ-Information eines bestimmten Types besorgen. Eine andere Möglichkeit wäre es, die Typ-Information in dem Operatorbaum zu verteilen. Ein Operator, der gerade einen neuen Typ erstellt hat, könnte seine Nachfolger informieren, indem die Typ-Information des neuen Typs in serialisierter Form an alle Nachfolger sendet. Dies sollte natürlich geschehen, bevor eine Ausprägung des neu erstellten Types gesendet wird. Für die serialisierte Form der Typ-Informationen, könnten zum Beispiel XML-Schemas verwendet werden.

Für diese Arbeit konnte kein konkreter Nutzen der dynamischen Typisierung festgestellt werden. Für zukünftige Erweiterungen aber, insbesondere für Funktionalitäten, die typischerweise in einem DSMS vorkommen, ist die Möglichkeit einer dynamische Typisierung interessant. Insbesondere die Optimierungs-Komponenten eines Anfrageprozessor müssen zumindest intern bei der logischen Optimierung des Operatorgraphs sehr detailliertes Wissen darüber haben, welche Operatoren welche Arten von Objekte produzieren und ob ein Tausch oder eine Zusammenfassung möglich ist.

Die Annahme hier ist, dass das Resultat des Selektionsschrittes der Visualisierungspipeline Objekte mit statischem zur Laufzeit bekanntem Typ sind. Ob während des Selektionsschrittes dynamische Typ-Informationen zur Verfügung standen oder nicht, ist hier nicht von Belang. Von der Existenz eines statischen Typsystems wird jedoch ausgegangen.

3.2.7. Modellierungssprache

Die Notwendigkeit eine eigene Modellierungssprache einzuführen, lässt sich leicht an einem Beispiel verdeutlichen. Nehmen wir an, das Resultat des Filterungs-Schrittes ist ein Nexus-Strom, welcher die Temperaturwerte von Temperatursensoren enthält. Wir wollen nun den Temperaturwert auf einen Punkt mit einer bestimmten Farbe abbilden. Hierfür muss dem Operator bekannt sein, wo die Information der Farbe eines Punktes in der verwendeten Modellierungssprache zu finden ist, um diese zu ändern. Lässt man nun verschiedene Modellierungssprachen zu, muss ein Mapping-Operator jede dieser Modellierungssprachen unterstützen. Der in dieser Arbeit verfolgte Ansatz favorisiert ein einheitliches Datenmodell für alle Objekte, die während des Filterungs-Schrittes oder des Geometrieerzeugungs-Schrittes erzeugt werden. Damit müssen nicht verschiedene Serialisierer und Deserialsierer für das jeweilige Datenformat der Modellierungssprache entwickelt werden, sondern es kann eine Standard-Bibliothek, die allen Operatoren zur Verfügung steht, die Augmented-World-Bibliothek, verwendet werden.

Das Ergebnis der Geometrieerzeugung sind die Geometriedaten in Form von *VisualObjects*. VisualObjects wurden mit Hilfe eines erweiterten Klassenschema für NEXUS, dem *Visual-ClassSchema*, definiert (siehe 2.4.1). In Abbildung 3.8 ist ein Ausschnitt der Klassenhierarchie zu sehen. Im Folgenden sollen die inhaltlichen Aspekte der darin vorkommenden Klassen erläutert werden.

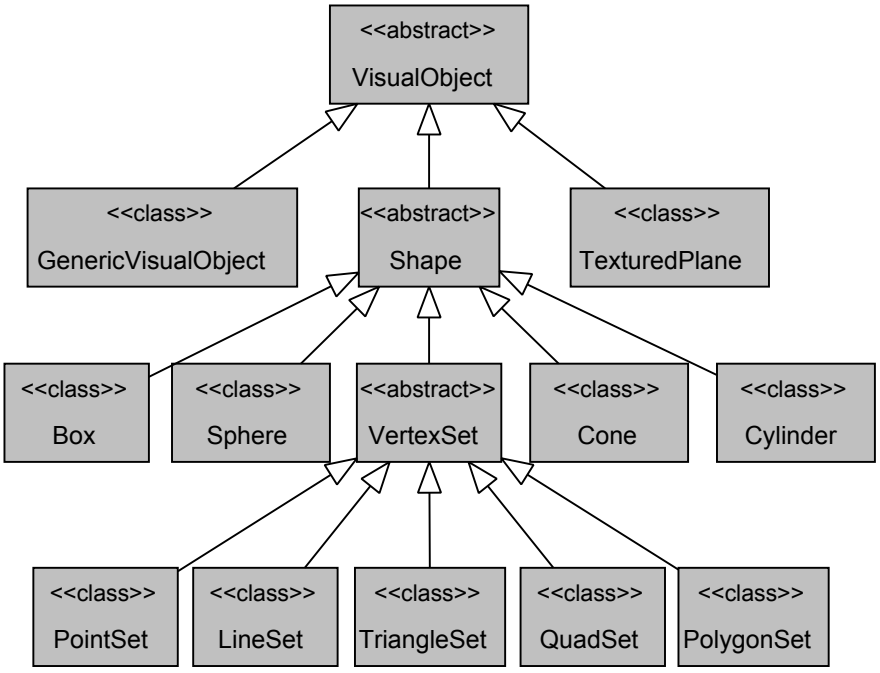


Abbildung 3.8.: Klassenhierarchie der VisualObjects

VisualObject

Das *VisualObject* ist die Superklasse aller visuellen Objekte, die die Rendering-Komponente anzeigen kann. Es enthält Attribute, die von allen Subklassen des VisualObject benötigt werden. Dies ist zum Beispiel die *Position* innerhalb des kartesischen Koordinatensystems der Rendering-Komponente oder ein Attribut, welches eine *Rotation* um eine beliebige Achse beschreibt. Zusätzlich werden hier Attribute definiert, die eine Interpolation zwischen Positionen ermöglicht, um ruckartige Positionsänderungen zu vermeiden. Um das visuelle Objekt global eindeutig referenzieren zu können, enthält es einen *Nexus Object Locator (NOL)*.[5]

GenericVisualObject

Mit Hilfe des *GenericVisualObject* kann eine externe Modellierungssprache verwendet werden, um das geometrische Modell zu definieren. Dies ist zum Beispiel sinnvoll, wenn das Modell sehr komplex ist und nicht mit den anderen Subklassen des VisualObjects auf effiziente Weise definiert werden könnte. Die Eigenschaften des Modells, das in der externen Modellierungssprache definiert wurde, bleiben transparent für die Rendering-Komponente. Damit ist es nicht möglich, Einfluss auf diese Eigenschaften zu nehmen. Beispielsweise kann die Farbe nicht geändert werden. Nur die Position oder die Rotation eines Objektes kann durch das Senden eines entsprechenden *UpdateCommand*s im Nachhinein noch verändert werden. Anwendung findet dies in dem Anwendungsszenario, bei dem es um die Visualisierung von mobilen Objekten geht (siehe 6.1).

Da die externe Modellierungssprache in einem beliebigen Format vorliegen kann, ist es möglicherweise nötig, das Modell in einer passenden Kodierung zu übermitteln. Die BASE64-Kodierung ([Lin93]) wäre eine Möglichkeit das Modell zu übermitteln, ohne dass vorhandene XML-Parser ersetzt werden müssten. Das *GenericVisualObject* enthält ein Attribut, das unter der Angabe einer konkreten Kodierung und der verwendeten Modellierungssprache mit Hilfe eines MIME-Typs [FB96] beliebige externe Daten transportieren kann (siehe Beispiel 3.2). Die Rendering-Komponente muss die Mechanismen zum Dekodieren und Laden eines Modells einer konkreten Modellierungssprache bereitstellen.[6]

```
<nsas:value>
       <eas:encoding>BASE64</eas:encoding>
       <eas:contentType>model/x3d+xml</eas:contentType>
       <eas:data>PD94bWwgdmVyc2lvbj0iMS4wIiBlb ... gk8L1NjZW5lPg0KPC9YM0Q+</eas:data>
</nsas:value>
```

Beispiel 3.2: Attributwert mit externen Daten

[5] Genau genommen enthält nicht *VisualObject* eine NOL, sondern eine Superklasse, die nicht in Abbildung 3.8 zu sehen ist (siehe hierzu Anhang A).

[6] Dies trifft momentan nur für X3D-Modelle [ISO04b] in einer BASE64-Kodierung [Lin93] zu.

TexturedPlane

Ein *TexturedPlane*-Objekt ist eine Viereck, auf dessen Vorderseite ein Bild gezeichnet wird. Für den Transport des Bildes, welches üblicherweise in einem binären Format vorliegt, wird ein auch hier ein Attribut verwendet, welches beliebige externe Daten transportieren kann (siehe Beispiel 3.2). Die Vorderseite des Vierecks ist als diejenige definiert, welche die Punkte in einer Reihenfolge entgegen des Uhrzeigersinns enthält. Ein *TexturedPlane*-Objekt könnte zum Beispiel verwendet werden, um als Hintergrund eine zweidimensionale Landkarte in Form eines Bildes darzustellen.

Shape

Shape ist die Superklasse aller geometrischen Formen. Es stellt eine Erweiterung des *VisualObject*s dar. *Shape* enthält ein Attribut mit mehreren Attributteilen, die das Aussehen der geometrischen Formen definieren. Sie definieren zum Beispiel die Farben, die Transparenz oder die Reflexionseigenschaften der geometrischen Formen, oder wie und ob die Kanten von Polygonen gezeichnet werden sollen.

Cone

Cone stellt einen Kegel dar. Er wird definiert durch die Angabe eines Radius und einer Höhe.

Sphere

Sphere stellt eine Kugel mit einem gewissen Radius dar.

Box

Box stellt einen Quader dar. Der Quader wird definiert durch die Angabe einer Höhe, Breite und Tiefe.

Cylinder

Cylinder stellt einen Zylinder dar, der durch eine Höhe und einen Radius definiert wird.

VertexSet

VertexSet ist die Superklasse von *PointSet*, *LineSet*, *TriangleSet*, *QuadSet* und *PolygonSet*. *VertexSet* erlaubt Definieren einer Menge von farbigen Punkten. Diese Menge wird von den Unterklassen jeweils unterschiedlich interpretiert.

PointSet

Das *PointSet*-Objekt stellt eine Menge von unabhängigen Punkten dar.

LineSet

Das *LineSet*-Objekt stellt eine Menge von Linien dar. Dabei gehören immer zwei aufeinanderfolgende Punkte zu einer Linie. Die Anzahl der Punkte muss gerade sein.

TriangleSet

Das *TriangleSet*-Objekt stellt eine Menge von Dreiecken dar. Ein Dreieck wird durch jeweils drei aufeinanderfolgende Punkte definiert. Die Vorderseite eines Dreiecks ist als diejenige definiert, welche die Punkte in einer Reihenfolge entgegen des Uhrzeigersinns enthält. Die Anzahl der Punkte muss durch drei teilbar sein.

QuadSet

Ein *QuadSet*-Objekt stellt eine Menge von Vierecken dar. Ein Viereck wird durch jeweils vier aufeinanderfolgende Punkte definiert. Die Vorderseite ist in analoger Weise zum Dreieck definiert. Die Anzahl der Punkte muss durch vier teilbar sein.

PolygonSet

Ein *PolygonSet*-Objekt stellt eine Menge von beliebigen Polygonen dar. Die Polygone können hier nicht, wie bei den vorangegangenen *VertexSet*-Erweiterungen, implizit definiert werden. Sie müssen explizit durch eine List von Zahlen angegeben werden. Dabei repräsentiert die n-te Zahl der Liste die Anzahl der Ecken des n-ten Polygons. Die Vorderseite wird auf die gleiche Weise wie die Vorderseite des Dreiecks oder Vierecks definiert.

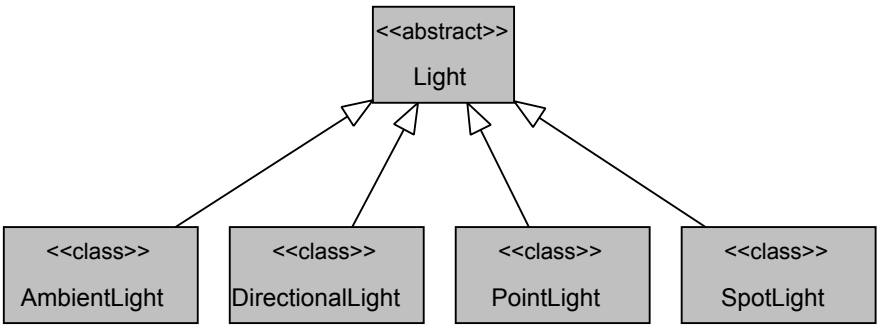


Abbildung 3.9.: Klassenhierarchie der Light-Klasse

Light

Eine gesonderte Rolle spielen Lichter. Sie sind keine Unterklasse von VisualObject, da es nicht für alle Licht-Objekte Sinn macht, eine Position oder Rotation zu modellieren. Die *Light*-Klasse is die Superklasse aller Lichter. Sie definiert die Farbe des Lichtes. Alle anderen Licht-Klassen erben die Farb-Information von dieser Klasse.

AmbientLight

AmbientLight stellt ambiente Beleuchtung dar. Dies ist eine Lichtart, welche Objekte von allen Seiten gleichmäßig beleuchtet.

DirectionalLight

Mit einem *DirectionalLight*-Objekt lassen sich parallele Lichtstrahlen modellieren. Neben des Farb-Attributs von *Light*, lässt sich eine Richtung angeben.

PointLight

PointLight ist eine Lichtquelle, die Licht von einer festen Position aus gleichmäßig in alle Richtungen ausstrahlt. Neben der Position der Lichtquelle kann die Art der Abschwächung des Lichts beeinflusst werden. Dabei gibt es einen konstanten Faktor und Faktoren, die die Abschwächung linear oder quadratisch zur Entfernung beeinflussen.

SpotLight

Mit Hilfe von *SpotLight* kann eine Lichtquelle simuliert werden, die von einer Position aus in eine bestimmte Richtung ausstrahlt. Neben Position und Richtung kann noch der Ausstrahlungswinkel und die Konzentration des Lichtes angegeben werden. Die Konzentration des Lichtes definiert die Abschwächung des Lichtes vom Zentrum zum Rand des durch den Ausstrahlungswinkel definierten Lichtkegels.

Einschränkungen

Wie in Kapitel 2.4.1 beschrieben ist das Datenmodell von NEXUS ein Objektmodell, welches allerdings auf eine Hierarchie von maximal zwei Attribut-Ebenen beschränkt ist. Inwiefern dies eine Einschränkung ist, lässt sich leicht an einem Beispiel erläutern. Nehmen wir an, wir wollen ein Haus modellieren. Die intuitive Vorgehensweise wäre hier nun, als Erdgeschoss einen Quader zu verwenden und als Dach eine Menge von Dreiecken. Damit müsste es möglich sein, dass ein visuelles Objekt mehrere Geometrien beinhalten kann. Jede dieser Geometrien kann nun komplex sein und wie das TriangleSet mehrere Punkten beinhalten. Jeder Punkt wiederum besteht aus einer Farbe und einer Koordinate, die die Position des Punktes angibt. Es wird klar, dass das eben genannte Beispiel mit dem aktuellen NEXUS-Datenmodell nicht möglich ist. Hierfür würde eine Hierarchie von mindestens drei Attribut-Ebenen benötigt. Die Entwicklung eines Datenmodells, welches eine beliebige Anzahl von Hierarchien beinhalten kann, ist ein mögliches Feld der Weiterentwicklung (siehe Kaptitel 7.2). Aktuell kann das Verbinden mehrerer Geometrien nur mit einer externen Modellierungssprache in Zusammenarbeit mit dem *GenericVisualObject* realisiert werden.

3.2.8. Anfragesprachen

Die grundlegende Frage bei der Definition einer Anfragesprache ist, in welchem Abstraktionsgrad diese Anfragesprache verfasst werden soll. Grundsätzlich lässt ein hoher Abstraktionsgrad von den Operator-Implementierungen die Freiheit, die für Optimierungsszwecke benötigt wird. Die Anfragesprache ist dann vollkommen unabhängig von den verwendeten Implementierungen der Operatoren. Es ist sogar möglich, mehrere semantisch äquivalente Implementierungen mit unterschiedlichen Laufzeiteigenschaften vorzusehen. Jedoch müssen hierfür die unterstützten Operatoren in einer abstrakten Sprachspezifikation semantisch klassifiziert werden. Dies hat den Nachteil, dass nur Operatoren unterstützt werden können, die auch in der Sprachspezifikation vorhanden sind. Bei der Visualisierung bestehen jedoch viele spezielle Anforderungen, die nicht alle von einem Visualisierungssystem unterstützt werden können.

Demgegenüber steht ein Ansatz, der einen geringen Abstraktionsgrad von den Operatoren vorsieht. Eine typische Herangehensweise hierbei wäre es, dem Benutzer die Möglichkeit zu geben, die Eingänge und Ausgänge verschiedener von ihm ausgewählten Implementierungen von Operatoren miteinander zu verbinden. Dies ähnelt dem Boxen-und-Pfeil-Modell,

welches zum Beispiel von Aurora, einem DSMS, verwendet wird (siehe hierzu [ACc+03]). Da hier allerdings die Operator-Implementierungen frei auswählbar wären, und dem Anfragebearbeitungssystem die Semantik eines speziellen Operators womöglich unbekannt wäre, könnte der von dem Benutzer spezifizierte Operatorbaum nicht weiter optimiert werden. Der Anfrageprozessor wäre dann lediglich für die Aufteilung auf die physischen Knoten zuständig.

Ein Kompromiss wäre es, das System mit einer gewissen Menge an Basis-Operatoren auszustatten, das Verwenden von domänenspezifischen Operatoren aber zuzulassen. Die Basis-Operatoren könnten dann semantisch klassifiziert werden und Umformungsregeln für diese definiert werden. Damit wäre eine Optimierung der Basis-Operatoren möglich.

Die verschiedenen Anwendungsfelder, in denen Anfragesprachen zum Einsatz kommen, habe in der Regel Einfluss auf die konkrete Syntax und Möglichkeiten einer Anfragesprache. Beispielsweise könnte eine Anfragesprache zur Spezifikation von Anfragen, die Selektionen, Projektionen, Joins oder andere für ein DSMS typische Operatoren enthalten, an die Standardsprache der Datenbanken SQL angelehnt sein.[7] Hingegen für Visualisierungsanfragen, die prinzipiell einzelnen Schritte des Visualisierungsprozesses und deren Reihenfolge spezifizieren, wäre eventuell eine Syntax von Vorteil, die dem Boxen-und-Pfeil-Paradigma [ACc+03] entspricht und die es ermöglicht, benutzerdefinierte Operatoren auszuwählen.

Auch Optimierungskriterien können sich von Anwendungsdomäne zu Anwendungsdomäne unterscheiden. Während ein DSMS in Anbetracht mehrerer ähnlicher Anfragen von verschiedenen Clients typischerweise versucht, Teile des Operatorgraphen für mehrere Anfragen gemeinsam zu verwenden, kann diese Strategie bei einem Visualisierungssystem problematisch werden. Würden sich beispielsweise zwei Clients einen Operator für die Bildgenerierung teilen, so könnten durch unterschiedliche Navigationswünsche Konflikte entstehen. Das schließt das gemeinsame Nutzen von Teilen des Operatorgraphens nicht grundsätzlich aus. Es gelten lediglich andere Regeln für die gemeinsame Nutzung Operatoren. Deswegen entsteht der Bedarf an anwendungsspezifischen Anfrageprozessoren.

Deswegen soll in dieser Arbeit grundsätzlich davon ausgegangen werden, dass für jedes Anwendungsfeld eigene Anfragesprachen existieren können, die sich auch syntaktisch untereinander unterscheiden. Damit müsste für jede Anwendungsdomäne auch eine Subkomponente existieren, die eine Anfrage in der jeweiligen Anfragesprache interpretiert und die entsprechenden Operatoren auf den Knoten zur Ausführung bringt. In Abbildung 3.10 ist die Architektur einer solchen Konstellation für eine Visualisierungsanfrage zu sehen. Die visuelle Anfrage wird in der imaginären Anfragesprache Visual Augmented World Query Language (VAWSL) gestellt. Diese wird von dem VAWSL-Prozessor interpretiert und die entsprechenden Operatoren auf den Knoten zur Ausführung gebracht. Für den Selektions-Schritt könnte die imaginäre DAWSL-Anfragesprache (Data-dependend Augmented World Stream Language) verwendet werden, deren Expressivität der Anfragesprache eines DSMS entsprechen würde. Es bleibt noch festzustellen, dass in diesem Fall die DAWSL-Anfrage aus der VAWSL-Anfrage extrahierbar sein muss. Es bietet sich also an,

[7] Ein Beispiel hierfür ist die im Rahmen des STREAM-Projektes entwickelte Anfragesprache Continuous Query Language (CQL) [ABW03]

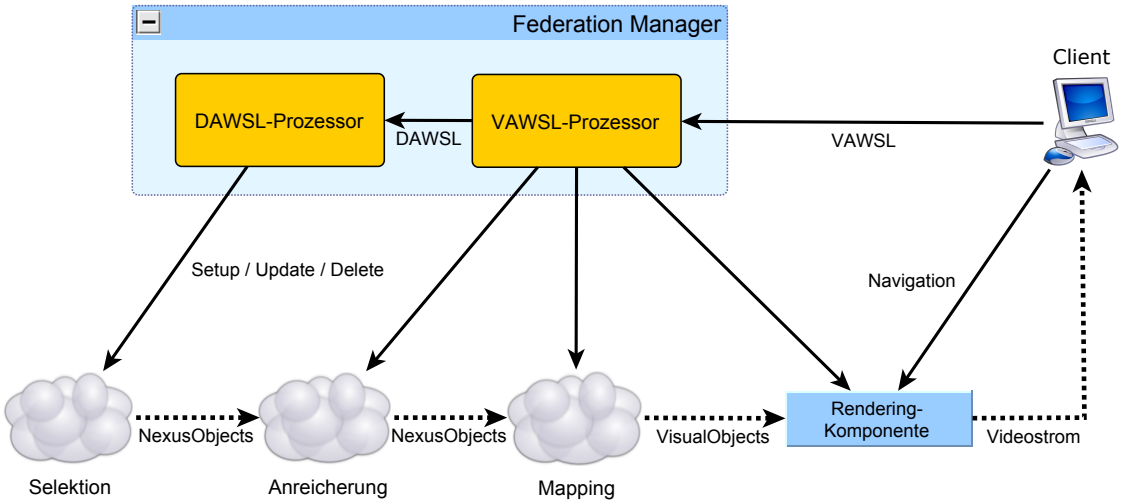


Abbildung 3.10.: Anfragesprachen unterschiedlicher Anwendungsfelder

die VAWSL-Anfragesprache so zu gestalten, dass für Definition des Selektionsschrittes die DAWSL-Anfragesprache verwendet werden kann.

Kapitel 4

Technologie

Es bietet sich an, für einige Teilbereiche dieser Arbeit auf schon vorhandene Technologie zurückzugreifen. Diese Teilbereiche lassen sich wie folgt eingrenzen:

1. Für die *Bildgenerierung* aus den abstrakten visuellen Objekten, die das Ergebnis des Geometrieerzeungs-Schrittes der Visualisierungspipeline sind, kann auf eine schon vorhandene Grafikbibliothek zurückgegriffen werden.
2. Das Versenden und Abspielen von Videoströmen an den Client, nachdem die Bildgenerierung stattgefunden hat, ist ein weiterer Teilbereich, für den sich eventuell verwendbare Bibliotheken finden lassen.

Die aktuelle Implementierung der Datenstrom-Föderation [Sar08] basiert auf einer Laufzeit-Umgebung für Java. Das schränkt die Wahl der zur Verfügung stehenden Technologien ein. Jedoch können an dieser Stelle nicht alle existierenden Technologien ausführlich diskutiert werden. Im Folgenden werden jeweils zwei Vertreter des jeweiligen Bereichs diskutiert und jeweils einer davon ausgewählt. Der Bereich der Bildgenerierung lässt sich zum Beispiel mit einer Grafikbibliothek wie Java OpenGL (JOGL) oder Java3D realisieren. Für das Versenden und Abspielen von Videoströmen könnte das Java Media Framework (JMF) oder die Quicktime-API dienen.

4.1. JOGL

Java OpenGL (JOGL) ist eine Grafikbibliothek, die auf der OpenGL-Bibliothek basiert. *OpenGL (Open Graphics Library)* ist eine softwareseitige Schnittstelle zur Grafik-Hardware. Aus Sicht des Anwendungsprogrammierers ist OpenGL eine Bibliothek, die aus einer Sammlung von Funktionen und Prozeduren besteht, die verwendet werden können, um grafische Objekte in einem zwei- oder dreidimensionalen Koordinatensystem darstellen zu können. Es sind für diverse Plattformen Implementierungen der offenen Spezifikation von OpenGL [SA06] verfügbar. Um die Funktionen der nativen Bibliothek einer bestimmten Plattform aber in Java verfügbar zu machen, muss ein gewisser Übersetzungsschritt stattfinden. Um Funktionen der Bibliotheken des Betriebssystems aufzurufen, steht auf Java-Seite das *Java*

Native Interface (JNI) ([SM08c]) zur Verfügung. JOGL ist nun eine Sammlung von Methoden und Klassen, die die Funktionalität von OpenGL realisiert, indem intern die Aufrufe der Methoden über JNI an die entsprechende OpenGL-Implementierung weitergeleitet werden. Durch Einführung der weiteren Abstraktionsschicht sind Java-Programme, die JOGL verwenden, plattformunabhängig (unter der Voraussetzung, dass auf der Plattform eine OpenGL-Implementierung zur Verfügung steht).

4.2. Java3D

Java3D ist eine Schnittstelle zur Anwendungsprogrammierung (Application Programming Interface, API) von 3D-Anwendungen oder Applets. Java3D abstrahiert dabei von den Programmierschnittstellen bzw. Bibliotheken niederer Ebene, wie Direct3D [Mic08], OpenGL [SA06], QuickDraw3D [App08] oder XGL [Fre08]. Aus Leistungsgründen benutzt Java3D allerdings genau diese Bibliotheken für die Bildgenerierung. Dies hat zur Folge, dass Java3D, anders als andere in Java geschriebene APIs, nur bedingt plattformunabhängig ist. Denn eine dieser Bibliotheken muss auf der verwendeten Plattform vorhanden sein, um Java3D benutzen zu können.

Java3D bietet dem Anwendungsprogrammierer die Möglichkeit, ein virtuelles dreidimensionales Universum zu modellieren und dabei die virtuellen Objekte dieses Universums in dem so genannten *Szenengraphen (Scene Graph)* anzuordnen. Der Szenengraph ist eine baumartige Modellierung der Objekte des virtuellen Universums. In Abbildung 4.1 ist eine Beispielausprägung eines Szenengraphen zu sehen.

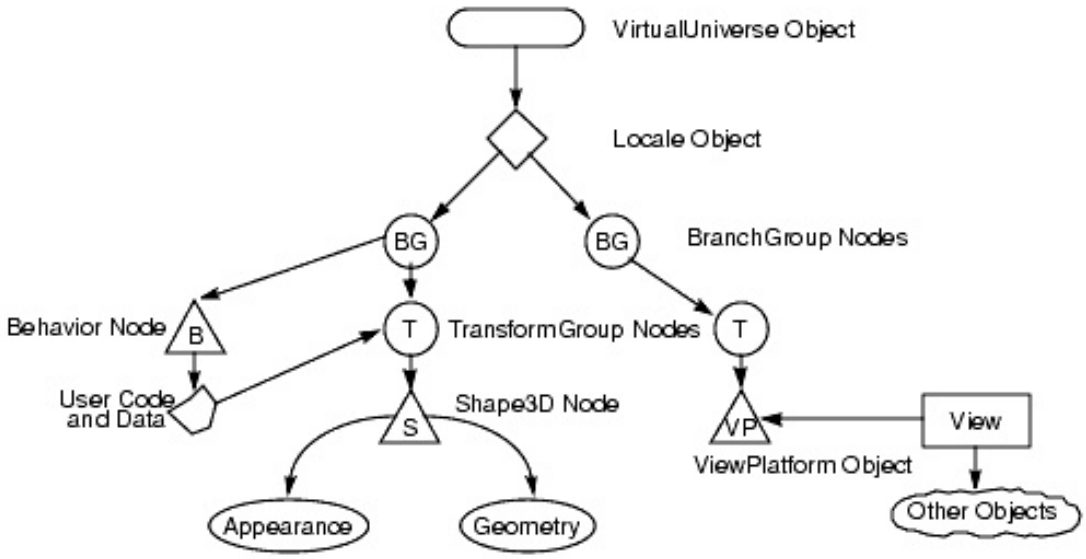


Abbildung 4.1.: Java3D Scene Graph [SRD00]

Die Wurzel des Baumes ist das *VirtualUniverse*-Objekt (virtuelles Universum). Es ist der zentrale Zugangspunkt der Anwendung zu dem virtuellen Universum. Ein virtuelles Universum kann mehrere *Locale*-Objekte enthalten. Das Locale-Objekt definiert den Ursprung des Koordinatensystems, das für alle darunter liegenden Objekte gilt. Unter einem Locale-Objekt können wiederum viele *BranchGroup*-Objekte angeordnet sein. Ein BranchGroup-Objekt stellt die Wurzel eines Teilgraphs des Szenengraphen dar. Innerhalb eines BranchGroup-Objekts können eine Vielzahl von anderen Knoten, zum Beispiel weitere BranchGroup-Objekte, angeordnet werden. Im Beispiel befindet sich in dem Teilgraphen auf der linken Seite ein *TransformGroup*-Objekt. Das TransformGroup-Objekt definiert die Position relativ zum Locale-Objekt, die Orientierung und die Größe aller geometrischen Objekte, welche sich unterhalb des TransformGroup-Objektes befinden. Im Beispiel befindet sich hier ein *Shape3D*-Knoten, der ein *Geometry*- und *Appearance*-Objekt referenziert. Das *Geometry*-Objekt definiert die geometrischen Eigenschaften des Objektes. Das *Appearance*-Objekt beinhaltet Eigenschaften, die das Aussehen betreffen, wie zum Beispiel Farbe, Reflexionseigenschaften des Lichtes, Transparenz oder viele andere. Die Geometrie und das Aussehen betreffenden Eigenschaften werden deshalb nicht direkt in dem Shape3D-Objekt modelliert, weil hierdurch ein gemeinsames Benutzen dieser Daten möglich wird. Um Objekte mit benutzerdefiniertem Verhalten zu versehen, können *Behavior*-Knoten eingefügt werden. Ein Behavior-Knoten enthält gewissen Programmtext, der an einer gewissen Stelle des Bildgenerierungsprozesses mit Hilfe einer Callback-Methode aufgerufen wird. Der Programmtext kann nun beliebige Knoten des Szenengraphs manipulieren. Damit könnte zum Beispiel die Position, bzw. Orientierung oder Größe eines Objektes abhängig von einer Benutzereingabe beeinflusst werden. Letztlich befindet sich auf der rechten Seite unterhalb einer TransformGroup ein *ViewingPlatform*-Objekt. Das durch die TransformGroup transformierte ViewingPlatform-Objekt definiert den Blickwinkel, den der Benutzer auf das virtuelle Universum hat. Das ViewingPlatform-Objekt wird von einem View-Objekt referenziert. Dieses enthält verschiedene Eigenschaften, die benötigt werden, um das Bild mit dem korrekten Blickwinkel zu generieren. Dies beinhaltet zum Beispiel den Monitor, das Fenster, in dem die Bildgenerierung stattfindet, oder Parameter, welche das physische Umfeld betreffen, wie die Augenposition des Benutzers.

Das objektorientierte Modell des Szenengraphen eignet für die Modellierung der heterogenen Objekte einer virtuellen oder realen Welt. Auch NEXUS verwendet eine objektorientierten Ansatz für die Modellierung von Kontextinformationen. Auf Grund der einfachen Übertragbarkeit der Elemente des NEXUS-Kontextmodells in das Modell des Szenengraphen wird für diese Arbeit Java3D als Grafikbibliothek verwendet.

4.3. Quicktime-Api

Die Multimedia-Plattform von Apple Inc. *QuickTime-API* [App05] ist für Windows- und Apple-Systeme verfügbar. Sie beinhaltet eine Vielzahl von Bibliotheken, einem eigenen Dateiformat und Anwendungen, wie den *QuickTime Player*. Neben dem reinen Abspielen von zeitbasierten Medien unterstützt die QuickTime API das Erstellen und Versenden von zeitbasierten Medien. Jedoch leider nicht für alle Betriebssysteme. Unter Windows können zeitbasierte Medien nur abgespielt werden, aber es können von Medien keine Datenströme

erstellt werden, die dann über ein Netzwerk gesendet werden. Beim Aufruf der entsprechenden Funktionen der QuickTime API meldet sich diese lediglich mit einer Ausnahme zurück, dass dieses für Windows-Plattformen nicht unterstützt wird. Der Grund hierfür ist, dass die QuickTime-API für das Codieren von Medienströmen spezielle Bibliotheken der Apple-Plattform benutzt. Dass Apple in Zukunft das Codieren von Medienströmen auch für Windows-Plattformen möglich macht, ist eher unwahrscheinlich, denn momentan sieht der Geschäftsfall hierfür nicht gewinnversprechend aus.

Für diese Arbeit bedeutet dies, dass für eine Verwendung der QuickTime-API auch mindestens ein Apple-Rechner zur Verfügung stehen müsste. Damit wäre es zwingend notwendig, dass der Knoten, der die Bildgenerierung vornimmt, auf einer Apple-Plattform basiert. Dies wäre zwar ohne weiteres möglich, würde jedoch die Freiheit, die zur Optimierung oder zur Lastverteilung zwischen den Knoten benötigt wird, einschränken. An dieser Stelle wird angenommen, dass die Anzahl der Apple-Rechner innerhalb der Datenstrom-Föderation in Zukunft eher gering ausfällt. Wie im vorangegangen Kapitel dargelegt, steht einem Client exklusiv eine Instanz eines RenderOperators zur Verfügung, die alle auf Apple-Rechnern ausgeführt werden müssten. Damit würde die Datenstrom-Föderation in anbetracht einer großen Anzahl von Clients nur mit der Anzahl verfügbaren Apple-Rechner skalieren. Deswegen wird von dieser Arbeit eher Abstand hiervon genommen, was jedoch nicht heißt, dass dies in Zukunft umgesetzt werden könnte. In der Tat gibt es in diesem Bereich noch viele Möglichkeiten zur Weiterentwicklung (siehe Kapitel 7.2).

4.4. JMF

Das von Sun Microsystems Inc. entwickelte *Java Media Framework (JMF)* [SM99] ist eine Programmierschnittstelle, die es ermöglicht, zeitbasierte Medien wie Audio oder Videoströme in Java-Anwendungen oder Applets zu integrieren. Was Java3D und JOGL in Bezug auf die Grafik-Hardware sind, ist JMF für Medien-Hardware, wie Kameras, Mikrofone und ähnliches. JMF besitzt eine 4-schichtige Architektur (siehe Abbildung 4.2). In der obersten Schicht befinden sich die Anwendungen oder Applets, die Entitäten der *JMF Presentation and Processing API (Präsentations- und Verarbeitungs-API)* verwenden. Diese wiederum werden realisiert von einer Menge austauschbarer *Plug-Ins* (Zusatzprogramme). Um die Präsentations- und Verarbeitungsschicht von den Plug-Ins zu isolieren und damit vor Änderungen in der Plug-In-Schicht zu bewahren, befindet sich zwischen den beiden Schichten die *JMF Plug-In API*. Die Plug-In-Schicht realisiert viele für die Medienverarbeitung wichtigen Teilaufgaben, wie das Encodieren und Decodieren von Audio- oder Videoströmen mit Hilfe von *Codecs*, oder das *Multiplexen* oder *Demultiplexen* zusammengesetzter Medienströme, wie zum Beispiel ein Videostrom mit Ton. Weiterhin können Medienströme durch *Effect*-Plug-Ins verändert werden, wie zum Beispiel die Erhöhung der tiefen Frequenzen in einem Audiostrom. *Renderer*-Plug-Ins werden dazu benötigt, aus den Daten der Medienströme eine Ausgabe unter Verwendung von Hardware, wie Grafikkarten oder Soundkarten, zu produzieren.

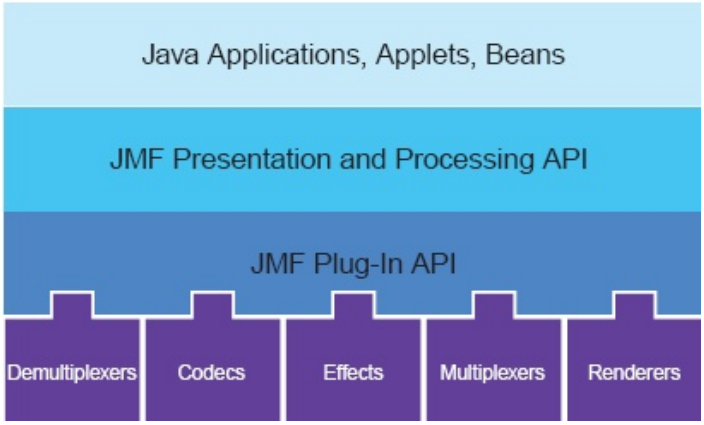


Abbildung 4.2.: JMF Architektur [SM99]

Das Plug-In-Konzept von JMF wird gestützt von einem Plug-In-Verzeichnis, dem *Plug-In-Registry* und dem *Plug-In-Manager*, der es erlaubt, Plug-Ins hinzuzufügen oder zu entfernen.

Die grundlegende Idee der Plug-In-Architektur war es wohl, die kosten- und zeitaufwendige Entwicklung von Codecs und Hardwareunterstützung durch das Plug-In-Konzept auf den Anwendungsentwickler abzuwälzen. Diese Strategie ist leider nicht aufgegangen. Denn Anwendungsentwickler vermeiden gerade deswegen die Verwendung von JMF, weil keine aktuellen Codecs zur Verfügung stehen, beziehungsweise weil die Hardwareunterstützung gering ist. Deswegen liegt die Weiterentwicklung von JMF derzeit auf Eis. Die letzten Änderungen erfolgten im Jahre 2004.

Damit scheidet JMF für diese Arbeit zumindest für die Generierung des Videostroms aus, denn für JMF stehen keine aktuellen Codecs zur Verfügung. Jedoch können Komponenten verwendet werden, welche das Abspielen von zeitbasierten Medien ermöglichen. Diese Komponenten sind unabhängig von den verwendeten Codecs und bieten auch Möglichkeiten, benutzerdefinierte Videoformate wiederzugeben.

JMF besitzt eine Implementierung zum Abspielen zeitbasierter Medien, das *JMStudio*. Teile dieser Implementierung können von Anwendungsprogrammierern verwendet werden, um die Medien in ihre Programme zu integrieren. In Abbildung 4.3 ist ein vereinfachtes Klassendiagramm der hierfür benötigten JMF-Komponenten zu sehen. Prinzipiell verarbeitet ein *Player* die Daten einer *Datenquelle (DataSource)*. Eine *DataSource* sollte neben einer Methode zum Starten und Stoppen auch die Möglichkeit anbieten, den Inhalt der Datenquelle zu beschreiben. Dies wird mit der `getContentType`-Methode ermöglicht, die den MIME-Typ der Daten zurückgibt, die die Datenquelle anbietet. Unter anderem gibt es in JMF *PullBufferDataSource*s und *PushBufferDataSource*s, welche wiederum eine beliebige Anzahl von *PullBufferDataStream*s bzw. *PushBufferDataSource*s verwalten. Der Anwendungsprogrammierer muss für das Verwenden der Player-Software eine der beiden *DataSource*s implementieren und mindestens einen *PullBufferDataStream* oder *PushBufferDataStream*. Die Entscheidung, ob die Pull- oder die Push-Variante gewählt wird, hängt vom jeweiligen Anwendungsfall ab.

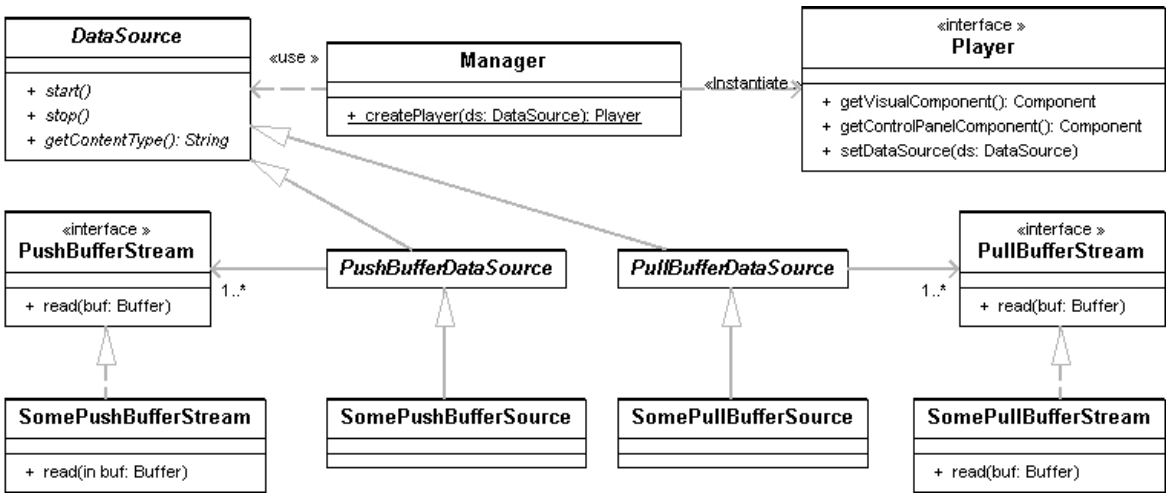


Abbildung 4.3.: Vereinfachtes Klassendiagramm der JMF-Komponenten

Der prinzipielle Unterschied zwischen den beiden Varianten ist die Semantik der Callback Methode `read(Buffer)`. Während im Pull-Fall diese Methode blockieren kann, wenn keine neuen Daten vorhanden sind, darf die Methode im Push-Fall auf keinen Fall blockieren.[1] Der read-Methode wird eine Referenz auf das `Buffer`-Objekt übergeben, in das die Daten eines Datenstroms hinein kopiert werden sollen. Die Buffer-Klasse ist ein vom jeweiligen Medienformat unabhängiger Datenbehälter, der die Möglichkeit bietet, gewisse Datenstücke mit einem Zeitstempel zu versehen und Format-Informationen zu versehen. Der Player kann nun mit Hilfe der Format-Information eventuell benötigte Codecs aus dem Plug-In-Verzeichnis laden und mit Hilfe der Zeitinformation aus dem Datenbehälter die Daten zum richtigen Zeitpunkt wiedergeben. Instanziert wird ein Player mit Hilfe des Managers. Der Player beinhaltet zudem noch Methoden, die `Component`-Instanzen der graphischen Komponenten zurückgeben. Diese können dann von den Anwendungen in ihre eigene graphische Oberfläche integriert werden.

[1] Die Begriffe „Pull“ und „Push“ sind von JMF-Seite etwas unglücklich gewählt. Üblicherweise beschreiben die beiden Begriffe die Beziehung eines Senders und Empfängers. Eine Push-Beziehung bedeutet, dass die Aktivität vom Sender ausgeht und der Empfänger passiv empfängt. Beim Pull-Ansatz hat der Empfänger die aktive Rolle und holt sich die Daten bei Bedarf vom Sender. Dies hat nichts mit dem hier beschriebenen Blockierverhalten gewisser Methoden zu tun.

Kapitel 5

Implementierung

Dieses Kapitel beschreibt die grundlegenden Konzepte der wichtigsten in dieser Arbeit entwickelten Komponenten. Diese können verwendet werden, um eine verteilte, virtualisierte Visualisierungspipeline zu realisieren. Es müssen hierfür nicht alle hier beschriebenen Komponenten verwendet werden. Der konkrete Aufbau der Visualisierungspipeline hängt von dem Anwendungsfall ab. In Kapitel 6 werden die Ausprägungen zweier Visualisierungspipelines zur Realisierung zweier unterschiedlicher Anwendungsszenarien vorgestellt. Dort werden teilweise Komponenten verwendet, die in diesem Kapitel detailliert beschrieben werden.

5.1. Komponenten

Zur Realisierung der Visualisierungspipeline mit Hilfe der Datenstrom-Föderation werden verschiedene Komponenten benötigt, die im Folgenden erläutert werden. Anstatt die komplette Klassenhierarchie zu beschreiben, werden lediglich die für das Verständnis relevanten Teile erläutert. Um kenntlich zu machen, welche Klassen die `Operation`-Schnittstelle der Datenstrom-Föderation realisieren, werden die Klassen in den folgenden Abbildungen der Übersicht wegen mit einer direkten Realisierungs-Beziehung zu dieser Schnittstelle versehen. In der Regel wird diese Schnittstelle nur indirekt durch die Erweiterung der abstrakten Klasse `AbstractOperation` implementiert. Für die komplette Klassenhierarchie einer typischen `Operation`-Realisierung sei auf Kapitel 2.4.2 verwiesen. Auf ein Klassendiagramm wurde komplett verzichtet, wenn die Klassenhierarchie die einer typischen `Operation`-Realisierung entspricht.

5.1.1. CoordTransform-Operator

Für die Transformation von geographischen Koordinaten wird das Verfahren nach der transversalen Mercator-Methode verwendet. Hierbei treten einige Parameter auf, die hier erläutert werden. In Abbildung 5.1 werden die geographischen Bedeutungen der Parameter verdeutlicht. Die beiden Parameter *Ursprungsbreitengrad* und *Ursprungslängengrad* legen

den eigentlichen Ursprung der Projektion fest. Der *Ursprungslängengrad*-Parameter, auch *Zentralmeridian*, genannt, ist genau der Längengrad, an dem der Zylinder der Projektion die Erdoberfläche, bzw. das zugrundeliegende Ellipsoid berührt. Die Parameter *False-Easting* und *False-Northing* legen einen Offset vom eigentlichen Ursprung fest. Dies ist notwendig, um bei Kartenprojektionen den Ursprung der Karte zum Beispiel auf die linke, untere Ecke zu verlegen. Dadurch werden negative Koordinatenangaben vermieden. Letztlich gibt es noch einen in der Abbildung nicht erwähnten Parameter, den *Skalierungsfaktor*. Betrachtet man in der Abbildung 5.1 den *Ursprungsbreitengrad*, so fällt auf, dass am Rand der Kartenprojektion erhebliche Fehler entstehen. Der Ort, an dem diese Ungenauigkeiten entstehen, lässt sich durch den Skalierungsfaktor beeinflussen. Der Skalierungsfaktor wird mit der X-Koordinate der projizierten Koordinaten multipliziert. In der Regel wird dieser kleiner gleich 1 gewählt und damit entstehen rechts und links neben dem Zentralmeridian zwei längentreue Längengrade. Bildlich vorgestellt bewirkt ein Skalierungsfaktor kleiner als 1, dass der Zylinder die Erdoberfläche nicht mehr nur berührt, sondern diese schneidet.

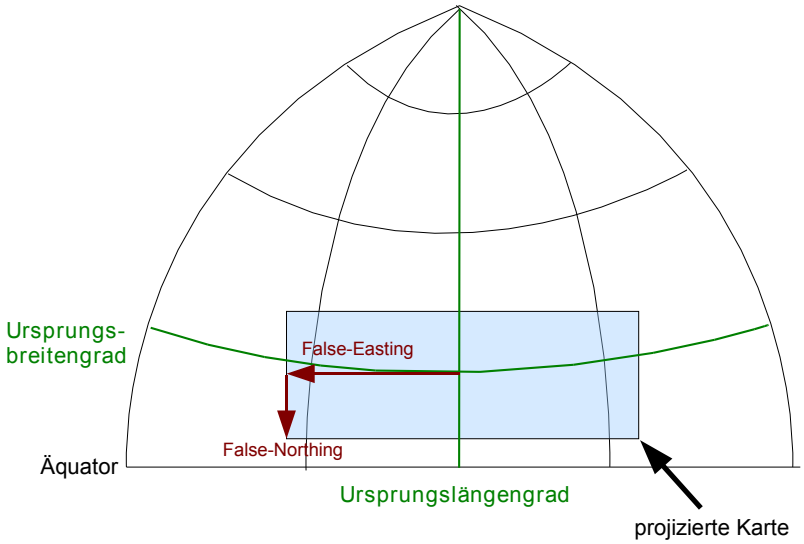


Abbildung 5.1.: Gauß-Krüger Projektion

Funktionsweise

Der `CoordTransform`-Operator realisiert eine Koordinatentransformation in ein projiziertes Koordinatensystem. Für die Durchführung der Transformation wurden Klassen der NEXUS *Geo-Classes*-Bibliothek verwendet. Mit Hilfe der Methode `setProperty(String, Object)` können dem Operator bei der Initialisierung Parameter übergeben werden. Durch einen Parameter kann bei der Initialisierung des Operators eine Implementierung von `CProjection` ausgewählt werden.[1] Ein weiterer Parameter spezifiziert das Attribut der eingehenden Objekte, in dem sich die zu transformierenden Koordinaten befinden. Der InputManager

[1] Momentan wird nur die Gauß-Krüger-Projektion nach dem transversalen Mercator-Verfahren unterstützt.

(siehe 2.4.2) wird irgendwann ein ankommendes Objekt aus der Warteschlange eines bestimmten Eingangs herausnehmen und mit Hilfe der Methode `process(int, Object)` an den Operator übergeben. Die zu transformierenden Koordinaten des Objektes werden mit Hilfe einer Instanz von `CProjection` transformiert und das Objekt weitergeleitet.

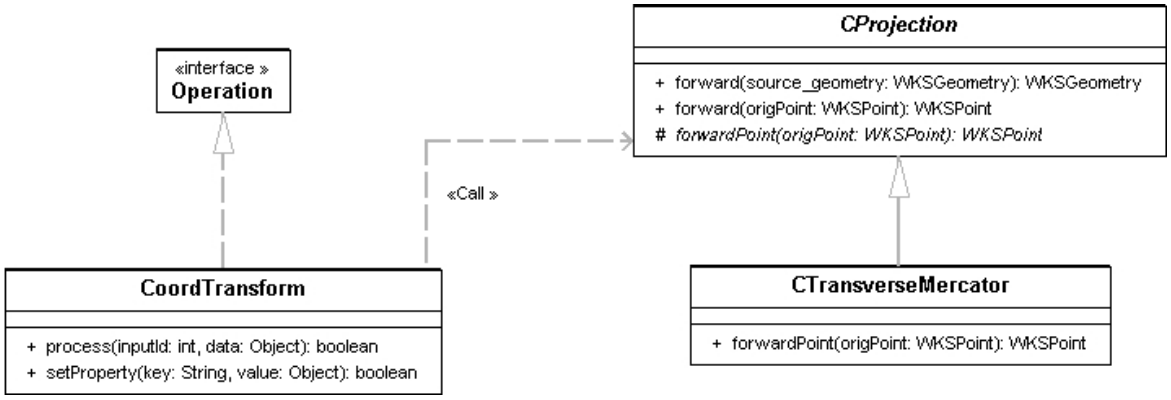


Abbildung 5.2.: Klassendiagramm des CoordTransform-Operators

`CProjection` ist die abstrakte Superklasse aller Projektionsverfahren. Sie implementiert die öffentlichen Methoden `forward(WKSGeometry)` und `forward(WKSPoint)`, welche die Transformation eines Punktes oder einer komplexen Geometrie, die aus mehreren Punkten bestehen kann, vornehmen. `WKSPoint` ist ein unabhängiger Datencontainer, der dazu verwendet wird, Tupel mit einer X-, Y- und Z-Koordinate zu modellieren. Die Klasse kann die Koordinaten eines beliebigen Koordinatensystems beschreiben. `WKSGeometry` ist die Oberklasse aller im NSAT vorkommenden Geometrietypen.[2] Die beiden öffentlichen `forward`-Methoden benutzen für die konkrete Transformation die Implementierung der `fowardPoint`-Methode einer Subklasse.

5.1.2. NAVGInterpolator-Operator

Der `NAVGInterpolator`-Operator realisiert ein Interpolationsverfahren, das auf zerstreuten Daten operiert. Hintergrund hierzu ist zum Beispiel die Untersuchung eines geographischen Gebietes mit Hilfe von Sensoren. Die Sensoren sind über die ganze zu untersuchende Fläche unregelmäßig verteilt. Jeder dieser Sensoren sendet in regelmäßigen Abständen einen skalaren Sensorwert. Diese Werte sind fest mit einer Position verbunden, die sich für jeden Sensor über die Zeit nicht ändert. Ziel ist es nun, zusätzlich zu den gemessenen Werten, virtuelle Werte zu berechnen. Die virtuellen Werte liegen auf einem imaginären Gitter. Die Gitter- und Zellengröße des Gitters kann mit Hilfe von Parametern festgelegt werden. Jeder virtuelle Wert wird berechnet, indem der Durchschnitt der N nächsten Datenpunkte gebildet wird, wobei N ein frei wählbarer Parameter ist. Abbildung 5.3 zeigt ein Gitter, in dem

[2] Dies ist nicht zu verwechseln mit den Objekten des *VisualClassSchema*, welche auch Geometrien enthalten. Die Geometrietypen des NSAT-Schemas werden von allen NEXUS-Anwendungen unterstützt, während die Objekte des *VisualClassSchemas* nur von den Komponenten der Visualisierungspipeline unterstützt werden.

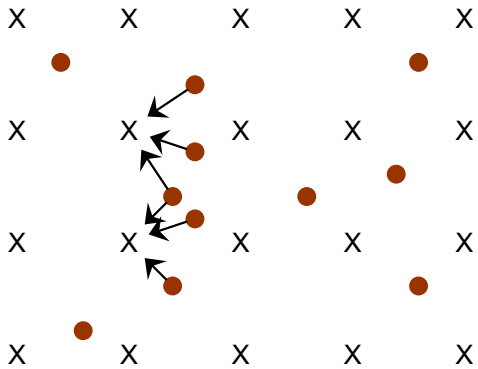


Abbildung 5.3.: Gitter der zu interpolierenden Gitterpunkte

zwei Gitterpunkte aus den jeweils drei nächsten Datenpunkten berechnet werden. Die Gitterpunkte werden als Kreuze, die Datenpunkte als Punkte dargestellt. Die Berechnungen der übrigen Gitterpunkte wurden der Übersicht wegen weggelassen. Von den Werten der Datenpunkte wird entweder der ungewichtete Durchschnitt gebildet, oder es wird ein gewichteter Durchschnitt berechnet, indem die Abstände der Datenpunkte vom Gitterpunkt invers proportional einfließen. Damit beeinflusst ein Datenpunkt, der weiter von einem Gitterpunkt entfernt ist, diesen entsprechend weniger. Vom Prinzip her entspricht dies der Shepard-Interpolation [She68], jedoch werden hier nicht alle Datenpunkte miteinbezogen, sondern nur die jeweils N nächsten. Formal ausgedrückt, berechnet sich der Durchschnitt $D(g_i)$ des Gitterpunts g_i wie folgt:

$$D(g_i) = \frac{\sum_{j=1}^{N} \frac{1}{dist(g_i, d_{ij})} \cdot v_{d_{ij}}}{\sum_{j=1}^{N} \frac{1}{dist(g_i, d_{ij})}}$$

Dabei berechnet $dist(g_i, d_{ij})$ den Abstand des Datenpunktes d_{ij} von dem Gitterpunkt g_i, und $v_{d_{ij}}$ stellt den skalaren Wert des Datenpunktes d_{ij} dar. Die Datenpunkte $d_{i1}, d_{i2}, ..., d_{iN}$ seien dabei die N nächsten Nachbarn eines Gitterpunktes g_i.

Das grundsätzliche Problem einer Interpolationsmethode, die den Datenwert des Gitterpunktes aus den Datenwerten aller Datenpunkte berechnet, ist der hohe Aufwand zur Aktualisierung der interpolierten Gitterpunkte. Bei der Ankunft jedes neuen Sensorwertes müsste für jeden Gitterpunkt ein neuer Wert interpoliert werden. Die Anzahl der Datenpunkte zu beschränken, die einen Gitterpunkt beeinflussen, hat den Vorteil, dass die Aktualisierungsrate der Gitterpunkte durchschnittlich geringer ausfällt. Jedenfalls ist dies intuitiv zu erwarten. Es ist klar, dass dies für gewisse Verteilungen von Datenpunkten nicht gilt. Existieren beispielsweise nur 5 Datenpunkte und wurde $N = 5$ gewählt, so beeinflusst jeder

Datenpunkt jeden Gitterpunkt, was zu keiner Verkleinerung der Aktualisierungsrate führt. In Abschnitt 5.2.1 wurde die Aktualisierungsrate gemessen.

Funktionsweise

Die Sensorströme der verschiedenen Sensoren werden an irgendeiner Stelle vereinigt und an den `NAVGInterpolator` gesendet. Dieser empfängt die Werte aller Sensoren und speichert den jeweils aktuellen Wert aller Sensoren in seinem lokalen Cache. Es existiert für jeden Gitterpunkt eine Liste, die die N nächsten Datenpunkte in einer nach dem Abstand zum Gitterpunkt aufsteigenden Reihenfolge enthält. Sind noch nicht mehr als N Datenpunkte empfangen, so kann diese Liste auch weniger als N Datenpunkte enthalten. Für jeden ankommenden Datenwert wird überprüft, ob der Datenwert einen der Gitterpunkte beeinflusst. Hierfür muss in der Liste jedes Gitterpunktes überprüft werden, ob der ankommende Datenwert den Gitterpunkt beeinflussen würde. Dafür muss lediglich der Abstand des letzten Datenpunktes in der Liste zum Gitterpunkt mit dem Abstand des aktuell ankommenden Datenpunktes zum Gitterpunkt verglichen werden. Ist der Abstand größer, wird mit dem nächsten Gitterpunkt fortgefahren. Ist der Abstand geringer, so wird der letzte Datenpunkt aus der Liste entfernt und der ankommende Datenpunkt in die Liste aufgenommen. Anschließend wird für den Gitterpunkt der Durchschnitt neu berechnet und der so interpolierte Wert an die nachfolgenden Komponenten gesendet.

5.1.3. ExpressionMapper-Operator

Typischerweise findet in einer Visualisierungspipeline nach der Filterung der Daten die Geometrieerzeugung statt. Prinzipiell ist die Geometrieerzeugung eine Abbildung von beliebigen Daten (Instanzen von *NexusObject*) auf geometrische Darstellungen (Instanzen von *VisualObject*). Weil *VisualObjects* auf dem gleichen Datenmodell wie *NexusObjects* basieren, lässt sich eine Abbildung leicht deskriptiv formulieren. Aus diesem Grund wurde der `ExpressionMapper` entwickelt, der es erlaubt, eingehende Daten-Objekte unter der Angabe einer deskriptiven Abbildungsvorschrift auf andere Daten-Objekte abzubilden. Da *VisualObjects* eine spezielle Art der Daten-Objekte sind, lässt sich mit diesem Operator die Geometrieerzeugung realisieren.

Funktionsweise

Die prinzipielle Funktionsweise ist in Abbildung 5.4 zu erkennen. Dem Operator werden bei der Initialisierung zwei Objekte übergeben, die er fortan in seinem Cache hält: das *MappingRule*-Objekt und das *MapTo*-Objekt. Das erste enthält die Abbildungsvorschrift für den Abbildungsvorgang, das zweite fungiert als eine Schablone, in die die dynamischen Informationen der Abbildung kopiert werden. Im Beispiel enthält die Schablone statische Eigenschaften wie den Typen des Objektes oder die Farbe. Der Radius, welcher Teil der

dynamischen Abbildung ist, wird entsprechend des in der Abbildungsvorschrift enthaltenen Ausdrucks berechnet.

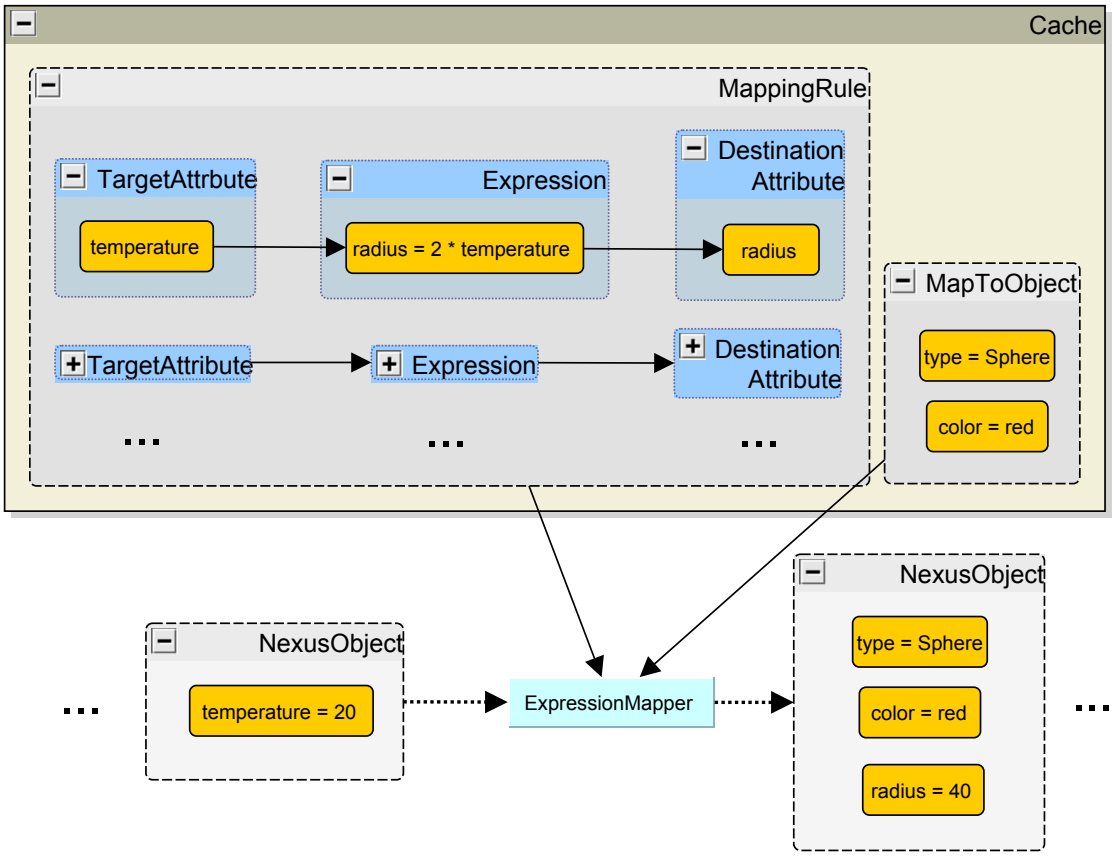


Abbildung 5.4.: Funktionsweise des ExpressionMapper-Operators

5.1.4. Bildgenerierungs-Komponenten

Während der Bildgenerierung werden aus den visuellen Objekten (*VisualObjects*) Bilddaten erstellt, die betrachtet werden können. Wie in Kapitel 3 dargelegt, kann die Bildgenerierung entweder auf dem Client stattfinden, oder von dem Visualisierungsnetzwerk durchgeführt werden. Im ersten Fall muss die `Sink<E>`-Schnittstelle implementiert werden im zweiten Fall die `Operation`-Schnittstelle. Um die gleiche Funktionalität nicht doppelt zu implementieren, wurde die Bildgenerierungsfunktionalität in die `Java3DRenderer`-Klasse ausgelagert. Damit können die beiden Klassen `RenderSink` und `RenderOperator` für die Bildgenerierung die `Java3DRenderer`-Klasse benutzen.

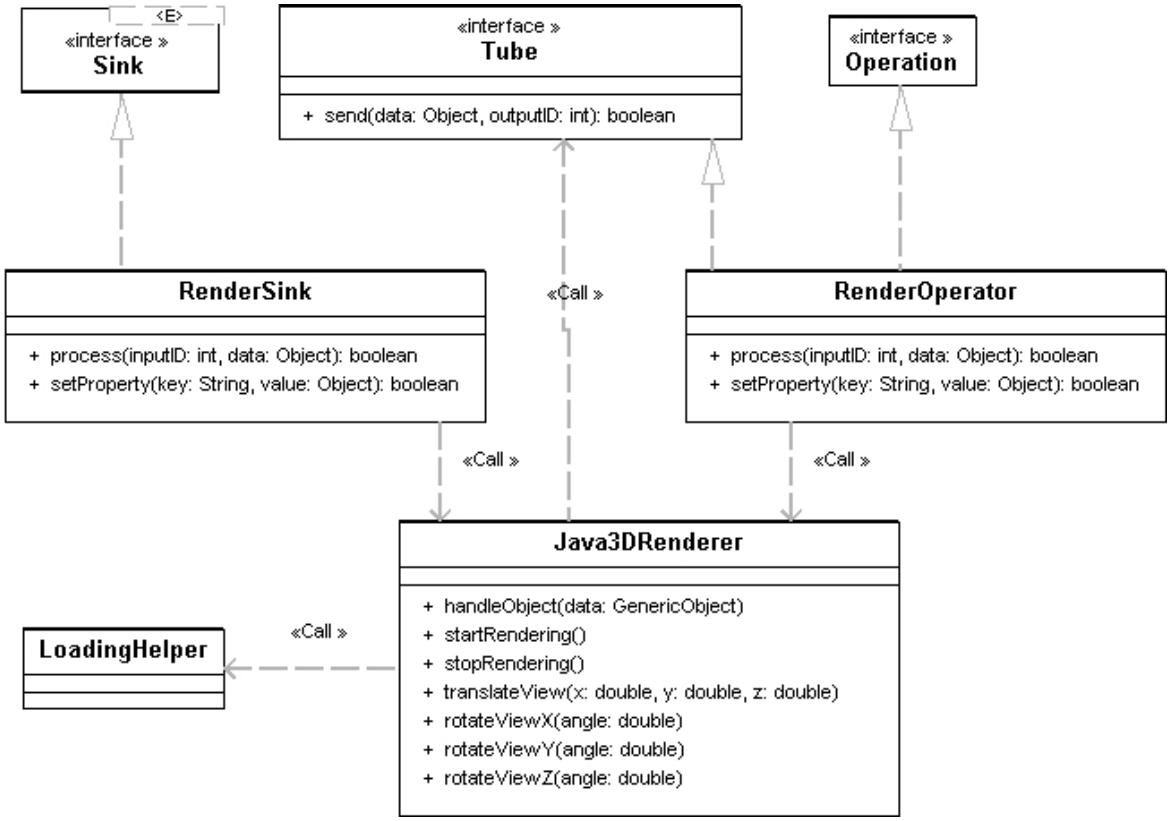


Abbildung 5.5.: Klassendiagramm Bildgenerierungs-Komponenten

Funktionsweise

Die Klassen `RenderOperator` und `RenderSink` instanzieren beide bei ihrer Initialisierung eine Instanz der `Java3DRenderer`-Klasse, die sie fortan für die Bildgenerierung benutzen. Wird der `Java3DRenderer` vom `RenderOperator` ausgeführt, so übergibt die `RenderOperator`-Instanz der `Java3DRenderer`-Instanz bei der Initialisierung eine Referenz auf sich selbst. Die `Java3DRenderer`-Klasse benutzt die Java3D-API für die eigentliche Bildgenerierung, die in einem eigenen Thread läuft. Fortan läuft der Rendering-Prozess und ein Fenster, das die Ausgabe des Rendering-Prozesses enthält, wird angezeigt. Wird nun ein Objekt empfangen, so ruft die zuständige `InputManager`-Instanz die Methode `process(int, Object)` auf. Sowohl der `RenderOperator` als auch das `RenderSink` delegieren diesen Aufruf an die Methode `handleObjekt(GenericObject)` der `Java3DRenderer`-Instanz, falls es sich beim empfangenen Objekt um eine Instanz von `GenericObject` handelt. Diese wiederum liest die in dem `GenericObject` enthaltenen Aktionen und reagiert entsprechend. Im einfachsten Fall hat das `GenericObject` den Typ *VisualObject*, und die `Java3DRenderer`-Instanz liest die in dem `GenericObject` enthaltenen Informationen des geometrischen Modells des darzustellenden Objektes mit Hilfe der statischen Methoden der `LoadingHelper`-Klasse und zeigt dieses an. Im anderen Fall hat das `GenericObject` den Typ *Command* und das entsprechende Kommando, wie das Entfernen eines Objektes von der Anzeigefläche, wird durchgeführt.

Wird die Bildgenerierung von dem `RenderOperator` ausgeführt, so sollen die erzeugten Bilder als Videostrom zu dem Client gesendet werden. Hierzu ist es in einem ersten Schritt notwendig, die erzeugten Bilder des Rendering-Prozesses an nachfolgende Operatoren oder Senken zu senden. Dafür verwendet der `Java3DRenderer` die `send(Object, int)`-Methode der `Tube`-Instanz.

Letztlich gibt es für den Client die Möglichkeit, Einfluss auf den Rendering-Prozess zu nehmen. Dies ist durch das dynamische Setzen von Parametern zur Laufzeit möglich. Werden vom Client Parameter gesetzt oder geändert, so wird von der Ausführungsumgebung die Methode `setProperty(String, Objekt)` aufgerufen. Aufrufe, welche den Blickwinkel *(View)* betreffen, werden an die entsprechenden Methoden des `Java3DRenderers`, wie `translateView` weitergeleitet. Eine weitere Art der Einflussnahme auf den Blickwinkel steht nur dem `RenderSink` zur Verfügung. Wurde der `Java3DRenderer` von diesem instanziert, so wird automatisch ein Tastatur-Navigator gestartet, der es ermöglicht, mit Hilfe der Pfeil- und Bild-Tasten innerhalb des visualisierten Szenarios zu navigieren.

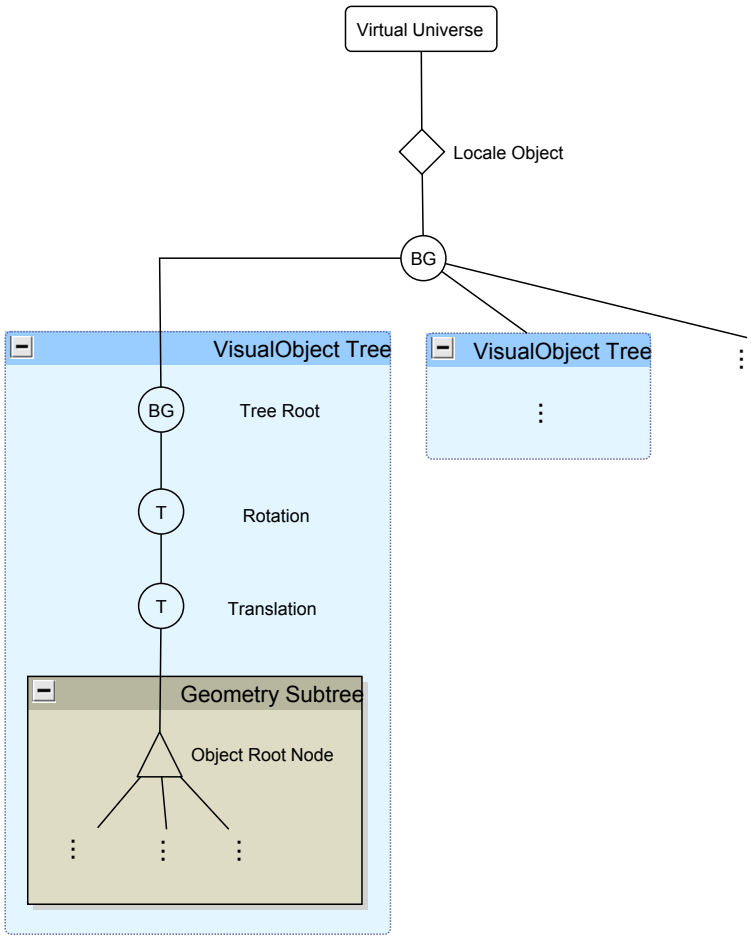


Abbildung 5.6.: Scenengraph des `Java3DRenderers`

Abbildung 5.6 zeigt den Szenengraphen der `Java3DRenderer`-Klasse. Jedes *VisualObject* wird beim Laden auf die baumartige Struktur der *VisualObject Tree* abgebildet. Diese enthält als Wurzel ein `BranchGroup`-Objekt, das ein `TransformGroup`-Objekt enthält, welches die Rotation eines Objektes steuert. Die darunter liegende `TransformGroup` definiert die Position eines Objektes. Die Position eines Objektes wird angesehen als die Translation des Objektes bezüglich des Ursprungs des kartesischen Koordinatensystems des `Locale`-Objektes. Die Struktur der *VisualObject Tree* ist fest definiert und dem `Java3DRenderer` bekannt. Dies ermöglicht es dem Java3DRenderer, gewisse Parameter, wie die Position zu verändern. Unterhalb der Translations-`TransformGroup` befindet sich der *Geometry Subtree*. Dieser enthält eine beliebige Knotenstruktur. Wurde das Objekt mit Hilfe der hier vorgeschlagenen Modellierungssprache modelliert, so ist dem `Java3DRenderer` auch die Struktur des *Geometry Subtree* bekannt und kann somit geändert werden. Es ist beispielsweise möglich, eine bestimmte Koordinate einer Unterklasse von *VertexSet* zu ändern. Wurde eine fremde Modellierungssprache im Zusammenhang mit dem *GenericVisualObject* verwendet, so ist dies nicht möglich. Die konkrete Struktur *Geometry Subtree* ist dann transparent für den `Java3DRenderer`, was es erst möglich macht, verschiedene, fremde Modellierungsprachen zuzulassen.

Eine gesonderte Rolle spielen *Light*-Objekte. Für sie werden keine baumartigen Strukturen gebildet, sondern sie werden lediglich unterhalb des *Root* `BranchGroup`-Objektes eingehängt.

5.1.5. Transport-Komponenten

Wenn Komponenten der verteilten Visualisierungspipeline auf verschiedenen Knoten ausgeführt werden, müssen die Objekte, die zwischen den Komponenten ausgetauscht werden, über das Netzwerk übertragen werden. Hierzu wurden die abstrakten Klassen `AbstractPlatformSink` und `AbstractPlatformSource` entwickelt. Zwischen einer Instanz von `AbstractPlatformSink` und `AbstractPlatformSource` besteht eine zuverlässige Verbindung.[3] Über diese Verbindung werden abgeschlossene Einheiten (*Rahmen*) in Form von Byte-Arrays übermittelt. Damit der Empfänger (`AbstractSimpleSource`) die Grenzen der Byte-Arrays erkennen kann, wird die Länge eines Byte-Arrays vor den eigentlichen Daten gesendet. Abbildung 5.7 zeigt das Format eines Rahmens.

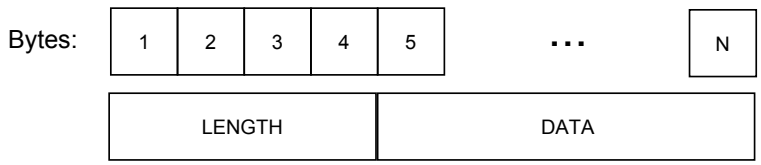


Abbildung 5.7.: Datenformat eines Rahmens

[3] Von einer *zuverlässigen Verbindung* spricht man in der Regel dann, wenn Übertragungsfehler, Datenverlust und Reihenfolgefehler ausgeschlossen sind (siehe hierzu u. a. [Tan03])

Funktionsweise

Die Klassenhierarchie des `AbstractPlatformSinks` ist in Abbildung 5.8 zu sehen. Das `AbstractPlatformSink` benutzt eine `NetworkGroup`-Instanz, um die für den Aufbau einer Verbindung benötigten Endpunkt-Informationen zu erlangen. Nach dem Aufbau einer Verbindung werden periodisch Objekte aus der Eingangswarteschlange entfernt. Die Objekte eines beliebigen Typs werden mit Hilfe der Implementierungen der abstrakten und generischen Methode `serialize(E data)` in Byte-Arrays serialisiert, welche mit Hilfe der Verbindung zur korrespondierenden Instanz einer `AbstractPlatformSource`-Implementierung versendet werden.

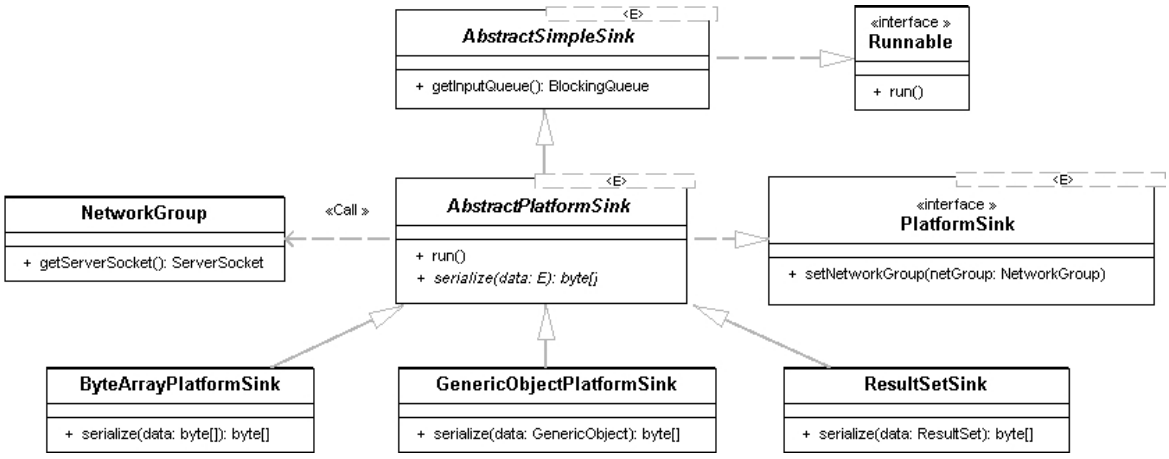


Abbildung 5.8.: Klassendiagramm des AbstractPlatformSink

Die Funktionsweise der `AbstractPlatformSource` und ihrer Implementierungen ist analog zu der oben beschriebenen, mit der Ausnahme, dass nun die Daten nicht mehr serialisiert, sondern deserialsiert werden und die Daten nicht mehr gesendet, sondern mit Hilfe einer `Socket`-Implementierung empfangen werden.

5.1.6. Videostrom-Verbindung zum Client

Wird die Bildgenerierung von dem `RenderOperator` durchgeführt, so ist eine Verbindung zum Client notwendig, um die Videodaten zu transportieren. Die Verbindung zum Client wird mit Hilfe eines Stroms von JPEG-Bildern realisiert. Für das Erstellen, Empfangen und Abspielen der JPEG-Bilder werden keine nativen Bibliotheken benötigt. Für den Client hat dies den Vorteil, dass dieser keine zusätzlichen Codecs benötigt, die möglicherweise auf der aktuellen Plattform nicht zur Verfügung stehen. Damit ist die Client-Software vollständig plattformunabhängig. Auf Client-Seite kann nun das Abspielen des empfangenen Videostroms von einer Senke oder einem Operator übernommen werden. Dies ist von der konkreten Anwendung abhängig. Soll eine direkte Kontrollleitung vom Client zum `RenderOperator` realisiert werden, um Interaktivität zu unterstützen, so muss ein Operator verwendet werden. Wird dies nicht benötigt, so kann auch eine Senke verwendet werden.

Abbildung 5.9 zeigt den Aufbau unter Verwendung einer Senke. Eine Kontrollleitung zum `RenderOperator` unter Verwendung eines Operators für das Abspielen des Videostroms wurde in Kapitel 6.1 realisiert.

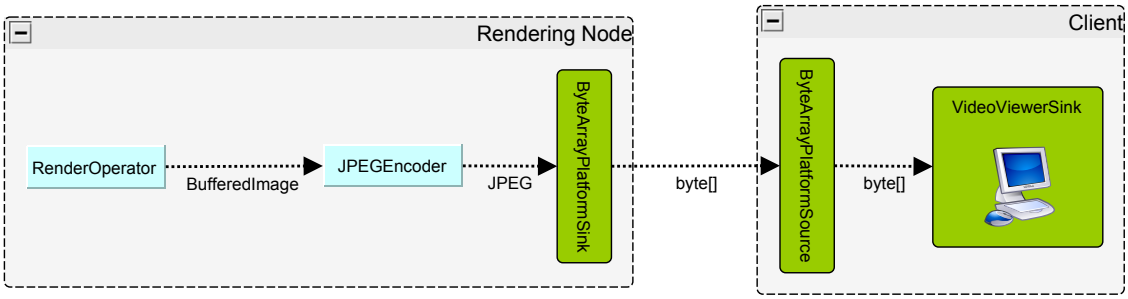


Abbildung 5.9.: Videostrom-Verbindung zum Client

In Abschnitt 5.1.5 wurde ein datenunabhängiges Übertragungsformat für Byte-Arrays definiert. Bei der Übertragung von Videoströmen werden zusätzlich typischerweise gewisse Frame-Informationen mitgesendet, die der abspielenden Software-Komponente mitteilen, wann ein empfangener Frame angezeigt werden soll. Der hier verfolgte Ansatz sieht es vor, eine Sequenznummer des Bildes mitzuliefern und einen Zeitstempel. Momentan wird für die Übertragung eine zuverlässige Verbindung benutzt, welche den fehler- und verlustfreien Empfang der Frames in der richtigen Reihenfolge garantiert. Infolgedessen, ist die Sequenznummer eigentlich überflüssig. Aber da im Zuge der Fortentwicklung eventuell unzuverlässige Verbindungen verwendet werden könnten, wurde die Sequenznummer in diesem Format zumindest schon einmal vorgesehen. Der Zeitstempel repräsentiert die absolute beim Sender vergangene Zeit in Nano-Sekunden seit dem ersten Frame, der den Zeitstempel 0 trägt. In Abbildung 5.10 ist das Format eines Video-Frames zu sehen, der mit Hilfe eines Daten-Frames transportiert wird.

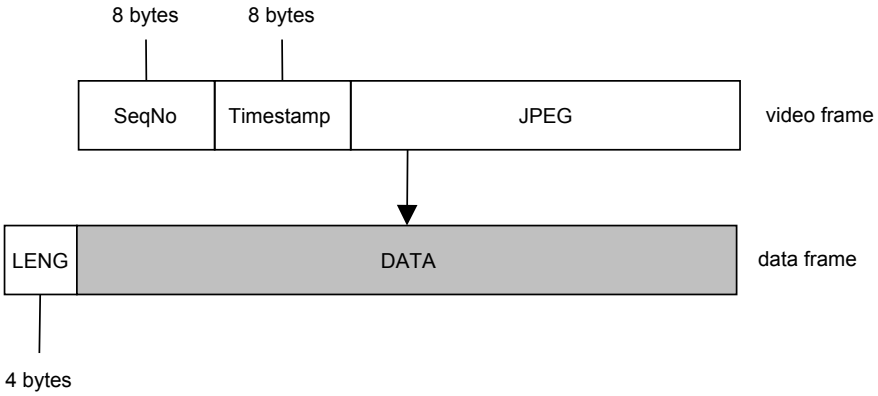


Abbildung 5.10.: Video-Frame und Daten-Frame

5.1.7. Funktionsweise

Wie in Abbildung 5.9 ersichtlich, sendet der `RenderOperator` Instanzen der `BufferedImage`-Klasse[4] an seinen Nachfolger, den `JPEGEncoder`. Dieser serialisiert die empfangen Rohdaten jedes Bildes in das JPEG-Format. Jedes Byte-Array, das von ihm an das `ByteArrayPlatformSink` weitergegeben wird, repräsentiert ein Bild, den so genannten *Frame*. Die Frames werden von dem `ByteArrayPlatformSink` über das Netzwerk übertragen und von der `ByteArrayPlatformSource` empfangen. Diese reicht die empfangenen Byte-Arrays an eine *VideoViewer*-Komponente weiter, die die Bilder auf dem Bildschirm wiedergibt.

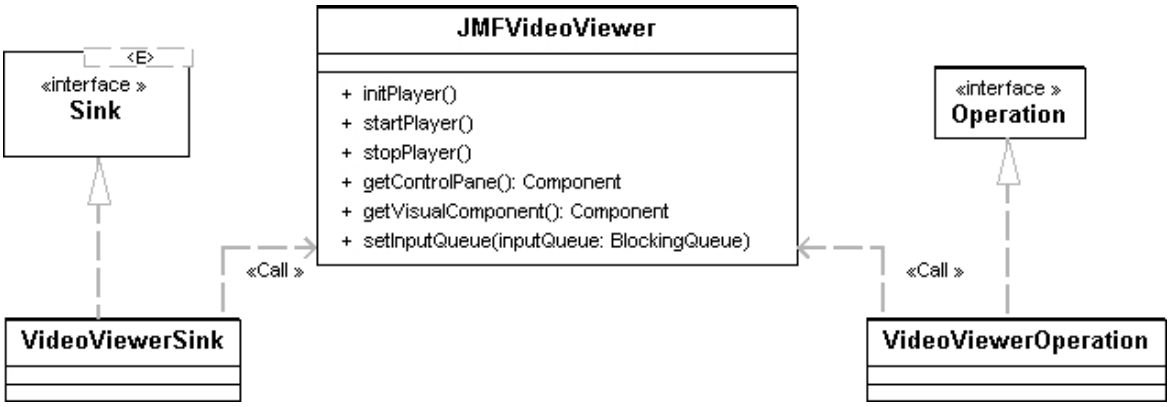


Abbildung 5.11.: Klassendiagramm der VideoViewer-Komponenten

Wie in Abbildung 5.11 wurde hier nach der gleichen Methode verfahren wie bei den Rendering-Komponenten. Die Funktionalität für das Abspielen der Videos wurde in die externe Klasse `JMFVideoViewer` ausgelagert, die nun sowohl von einer `Operation`-Implementierung als auch von einer `Sink<E>`-Implementierung genutzt werden kann. Die `JMFVideoViewer`-Instanz läuft in einem eigenständigen Thread. Die `Sink<E>`- oder `Operation`-Implementierung fügen empfangene JPEG-Bilder in eine Warteschlange ein, aus der die `JMFVideoViewer`-Instanz diese wieder entfernt. Die Warteschlange ist notwendig, weil sich die Ankunftsrate und die Abspielrate der JPEG-Bilder unterscheiden können. Dies ist zum Beispiel der Fall, wenn der Betrachter das Video in Zeitlupe betrachtet.

5.2. Evaluation

Die Evaluation hat das Ziel, die Tauglichkeit und Leistungsfähigkeit des hier entwickelten Prototyps zu testen. Im Folgenden werden strategische Messungen durchgeführt, die ein Bild auf die grundsätzlichen Leistungseigenschaften geben und helfen, die vorhandenen Flaschenhälse zu identifizieren.

[4] `BufferedImage` ist der zentrale Datenbehälter für Bilddaten der *AWT-API (Abstract Window Toolkit)*. *AWT* [SM08a] ist die Standard API für die Erstellung einer graphischen Benutzeroberfläche in Java

	System A	System B
Prozessor	AMD X2, 1,61 GHz	Intel Pentium M, 1,60 GHz
Speicher	2 GB RAM	1 GB RAM
Grafikkarte	NVIDIA GeForce Go 7600, 256 MB	ATI Mobility Radeon X700, 64 MB

Tabelle 5.1.: Verwendete Hardware-Systeme

Für die Tests wurden zwei Hardware-Umgebungen verwendet, die in Tabelle 5.1 spezifiziert werden. Für alle lokalen Messungen wurde die Hardware-Umgebung A verwendet. Das Betriebssystem beider Systeme ist Windows XP SP2. Es wurde in beiden Fällen die Java-JRE `1.5.0_08` verwendet.

5.2.1. Interpolation mit dem NAVGInterpolator

Während des Filtering-Schrittes der Visualisierungspipeline, kann eine Interpolation zwischen den Daten erfolgen. Die Interpolation kann zwischen beliebigen skalaren Werten, zum Beispiel zwischen Temperaturwerten von Thermometern, angewendet werden. In der Regel sind die zu interpolierenden Daten zerstreut. Mit dem NAVGInterpolator kann ein regelmäßiges Gitter aus Interpolationspunkten definiert werden, an denen jeweils ein neuer Datenwert interpoliert wird. Wenn in die Berechnung eines Interpolationspunktes alle Datenpunkte miteinfließen, so besteht das Problem, dass bei Ankunft eines neuen Datenpunktes, in diesem Fall also bei der Ankunft eines neuen Temperaturwertes, alle Interpolationspunkte neu berechnet werden müssen und die neu berechneten Werte den nachfolgenden Operatoren propagiert werden müssen. Der NAVGInterpolator realisiert eine näherungsweise Interpolation, indem nur eine feste Anzahl der nächstliegenden Datenpunkte in die Berechnung eines Interpolationspunktes miteinbezogen wird. Dies kann zu einer Verringerung der neu zu berechnenden Interpolationspunkte führen.

Hier wurde die Anzahl der neu zu berechnenden Interpolationspunkte bei der Ankunft eines neuen Datenpunktes untersucht. Diese Anzahl wird im Folgenden *Aktualisierungsrate* genannt. Abbildung 5.12 zeigt die Aktualisierungsrate zweier Versuchsläufe, in der Abbildung *Iterationen* genannt. Es wurden 100 zufällig verteilte Datenpunkte mit einer X- und Y-Koordinate zwischen 0 und 10 generiert. Entsprechend wurde ein Gitter mit jeweils zehn Gitterpunkten in X- und Y-Richtung definiert. Die Anzahl der in die Berechnung eines Interpolationspunktes einfließenden Datenpunkte betrug 3. Die zufällig generierten Datenpunkte werden dem NAVGInterpolator zweimal übergeben, wobei der Cache vor der ersten Iteration noch leer ist. Vor der zweiten Iteration befinden sich die Datenpunkte der ersten Iteration in dem Cache des Operators. Die zwei Iterationen sollen das periodische Senden von Sensorwerten simulieren.

Aus der Abbildung 5.12 ist klar eine Verringerung der Aktualisierungsrate in der ersten Iteration zu erkennen. Die ersten Datenpunkte führen noch zu keiner Veringerung der Aktualisierungsrate, da der ankommende Wert für jeden Interpolationspunkt, einen der

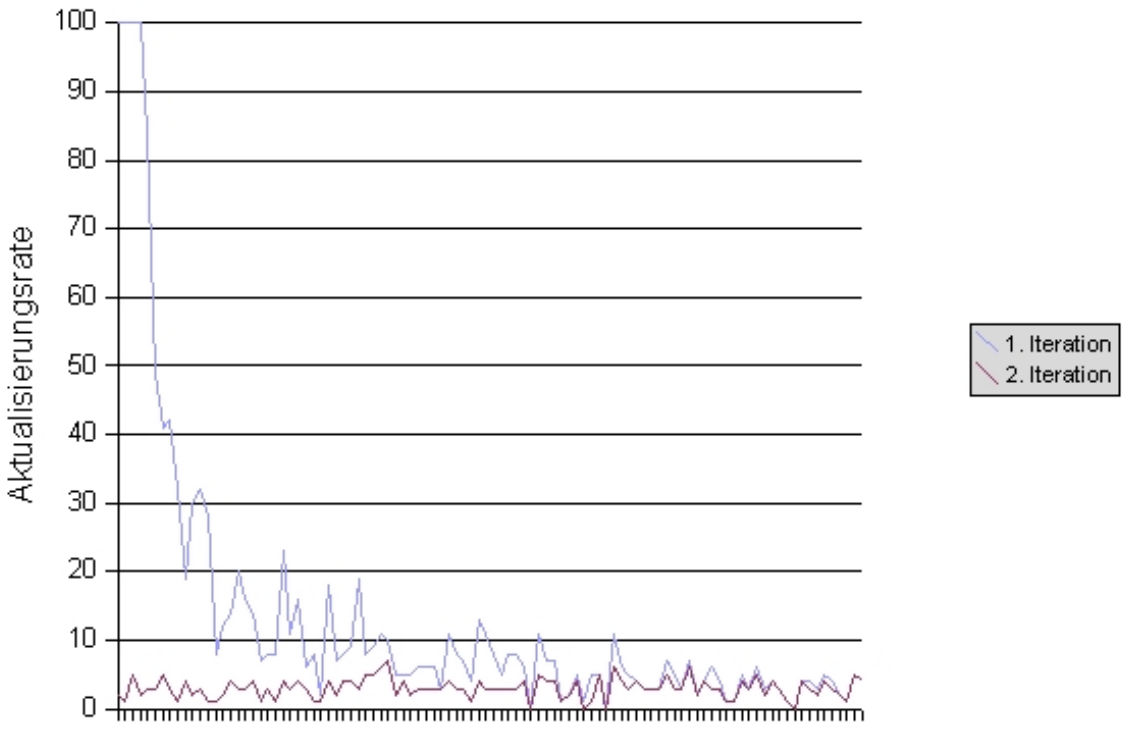


Abbildung 5.12.: Aktualisierungsrate des NAVGInterpolators

nächstliegenden Datenpunkte darstellt. Dann sinkt die Aktualisierungsrate rapide ab, bis sie sich der Aktualisierungsrate der 2. Iteration annähert. Das hat damit zu tun, dass sich der Cache des Operators langsam auffüllt und damit nicht mehr alle Datenpunkte in die Berechnung miteinbezogen werden müssen. Die Aktualisierungsrate der zweiten Iteration verharrt relativ stetig bei Werten von etwa drei Aktualisierungen pro Datenpunkt. Dies hängt von den hier gewählten Versuchbedingungen ab. Da es jeweils genau 100 Interpolations- und Datenpunkte gibt und in die Berechnung eines Interpolationspunktes die drei nächstgelegenen Datenpunkte miteinfließen, so ist es nicht verwunderlich, dass ein Datenpunkt durchschnittlich in die Berechnung dreier Interpolationspunkte miteinfließt.

Gezeigt werden konnte eine Reduzierung der Aktualisierungsrate. Die durchschnittlich zu erwartende Verarbeitungsdauer eines Datenpunktes ist jedoch mindestens linear zur Anzahl der Gitterpunkte. Die lineare Dauer ergibt sich aus der Tatsache, dass für jeden Gitterpunkt überprüft werden muss, ob der angekommene Datenpunkt, den interpolierten Wert beeinflussen würde. Das birgt noch Potenzial zur Verbesserung. Um sublineare Komplexität zu erreichen, könnten die Interpolationpunkte mit Hilfe eines räumlichen Indexes angeordnet werden.

5.2.2. Bildrate des Senders

Der Kern des Systems ist die Rendering-Komponente. Spätestens hier treffen alle Datenströme einer visuellen Anfrage zusammen, damit letztendlich ein Videostrom generiert werden kann. Ein weiterer potentieller Flaschenhals ist der hier verwendete proprietäre JPEG-Videostrom. Da die Bildgenerierung und die JPEG-Komprimierung für den hier entwickelten Prototyp auf dem gleichen Knoten stattfinden müssen, ist es angebracht, zu untersuchen, wie die Durchsätze dieser beiden Prozesse im Verhältnis zueinander stehen.

Das Objekt, von dem Bilder generiert werden, ist ein roter Würfel unter Einfluss einer gerichteten Lichtquelle, der sich in 4 Sekunden einmal um die eigene Achse dreht (siehe Abbildung 5.13).

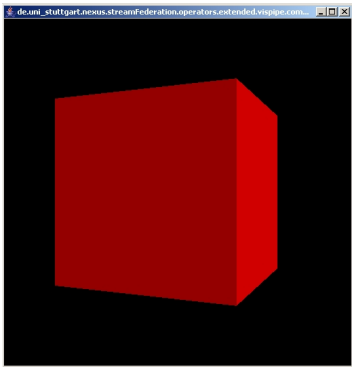


Abbildung 5.13.: Rotierender Würfel

Abbildung 5.14 zeigt die Bildraten der beiden unterschiedlichen Operatoren `RenderOperator` und `JPEGEncoder`. Der erste generiert die Bilder im Rohformat und der zweite komprimiert diese zu JPEG-Bildern. Die beiden Operatoren werden parallel auf demselben Knoten ausgeführt. Es wurden verschiedene Auflösungen untersucht. Als klar erkennbares Ergebnis, lässt sich ein Flaschenhals bei der JPEG-Komprimierung identifizieren. Ab einer Auflösung größer als 800×600 Pixel sinkt die Frame-Rate unter ein Niveau, das keine flüssige Darstellung des Videostroms mehr erlaubt. Auch die Frame-Rate des Rendering-Prozess sinkt bei höherer Auflösung leicht ab. Dies hat jedoch nichts mit der tatsächlichen Belastung der GPU[5] zu tun. Eine Ausführung ohne parallele JPEG-Komprimierung ergab, dass der Rendering-Prozess unabhängig von der Auflösung mit einer Rate von ca. 60 fps Bilder produziert, was wahrscheinlich mit der gewählten Monitor Frequenz zusammenhängt. Die Verringerung der Bildrate bei parallel laufender JPEG-Komprimierung hat mit der Belastung der CPU zu tun. Nach jedem Zyklus der Bildgenerierung wird die CPU benötigt, um das Bild aus dem Speicher der Grafikkarte in den Anwendungsspeicher zu kopieren. Wenn nun die CPU von der JPEG-Komprimierung gleichzeitig belastet wird, so benötigt dieser Kopiervorgang mehr Zeit.

Eine weitere Erkenntnis, die man aus der Abbildung gewinnen kann, ist die Tatsache, dass der Rendering-Prozess für ein großes Auflösungsspektrum mit einer höheren Rate läuft,

[5] Die *Graphics Processing Unit (GPU)* ist die zentrale Recheneinheit einer Grafikkarte.

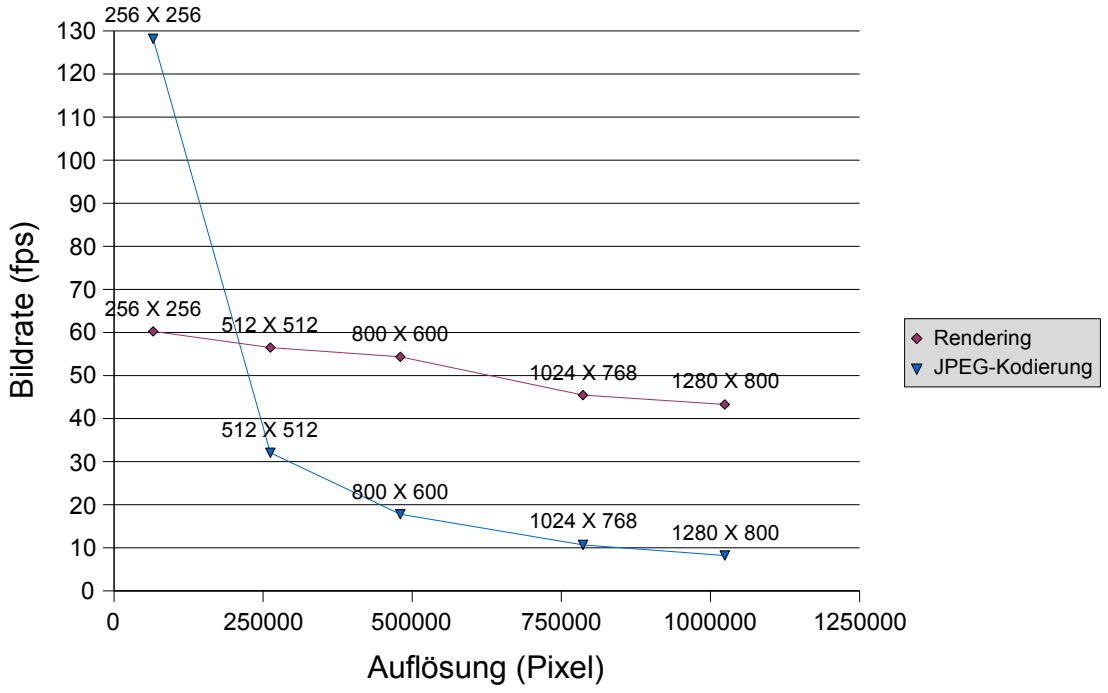


Abbildung 5.14.: Bildrate in Abhängigkeit von der Auflösung

als die JPEG-Komprimierung. Damit produziert der Rendering-Prozess schneller Bilder als diese komprimiert werden können, was ohne weitere Vorkehrungen zu Datenverlust führt. In diesem Fall läuft die Warteschlange, die sich zwischen dem Rendering-Prozess und dem Komprimierungs-Prozess befindet, über. Der Verlust von Bildern bei einer Bildrate von über 40 fps, ist kein gravierendes Problem, da dies dem (menschlichen) Betrachter nicht auffällt. Jedoch ist der Verlust eines Bildes aus Effizienzgründen ungünstig, denn für die Generierung des Bildes wurden umsonst Ressourcen in Anspruch genommen. Das Kernproblem ist das Verhalten der Ausführungsumgebung beim Senden eines Objektes. Momentan müssen Operatoren, um Objekte an ihre Nachfolger zu senden, die `send`-Methode der `Tube`-Schnittstelle verwenden. Diese blockiert nicht, falls die Warteschlange des Empfängers gerade voll ist, sondern kehrt in diesem Fall sofort zurück, ohne das entsprechende Objekt in die Warteschlange aufgenommen zu haben. Der Rendering-Prozess hat nun nur noch die Möglichkeit, eine unbestimmte Zeit zu warten und es noch einmal zu versuchen, oder das Bild zu verwerfen. Um die Latenz der folgenden Bilder nicht noch zu vergrößern, wurde hier der Ansatz des Wegwerfens gewählt. Für eine Bildgenerierung ohne Datenverlust muss momentan eine Auflösung kleiner als 512×512 Pixel gewählt werden, so dass die JPEG-Komprimierung schneller abläuft als der Rendering-Prozess. Dies birgt allerdings noch Potenzial zur Verbesserung, denn die JPEG-Komprimierung ermöglicht eine flüssige Darstellung des Videostroms bis zu einer Auflösung von ca. 800×600 Pixeln. Hierfür sollte die Ausführungsumgebung eine weitere Methode anbieten, die ein blockierendes Senden realisiert.

5.2.3. Instanzierungsdauer von VisualObjects

Die *VisualObject*s liegen im gleichen Datenmodell vor, wie alle anderen *NexusObjects*. Damit bleibt zu untersuchen, wie schnell aus dem generischen Datenobjekt `GenericObject`, welches das *VisualObject* enthält, eine Repräsentation gebildet werden kann, die die Rendering-Komponente anzeigen kann. Somit wird die Dauer des Übersetzungsschrittes von den Nexus-Datenobjekten in die Datenobjekte der Java3D-API gemessen. Der Rendering-Prozess wird in dem nächsten Zyklus der Bildgenerierung das frisch instanzierte Objekt anzeigen. Die Dauer eines Zyklus hängt von der Bildrate ab und beträgt beispielsweise bei einer Bildrate von 40 fps 25 ms.

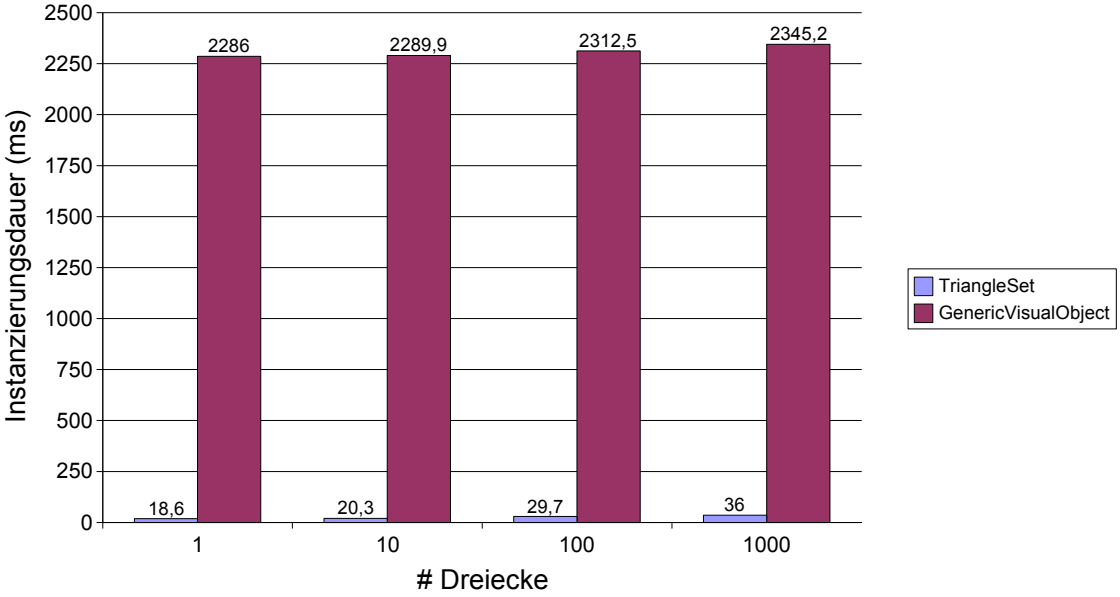


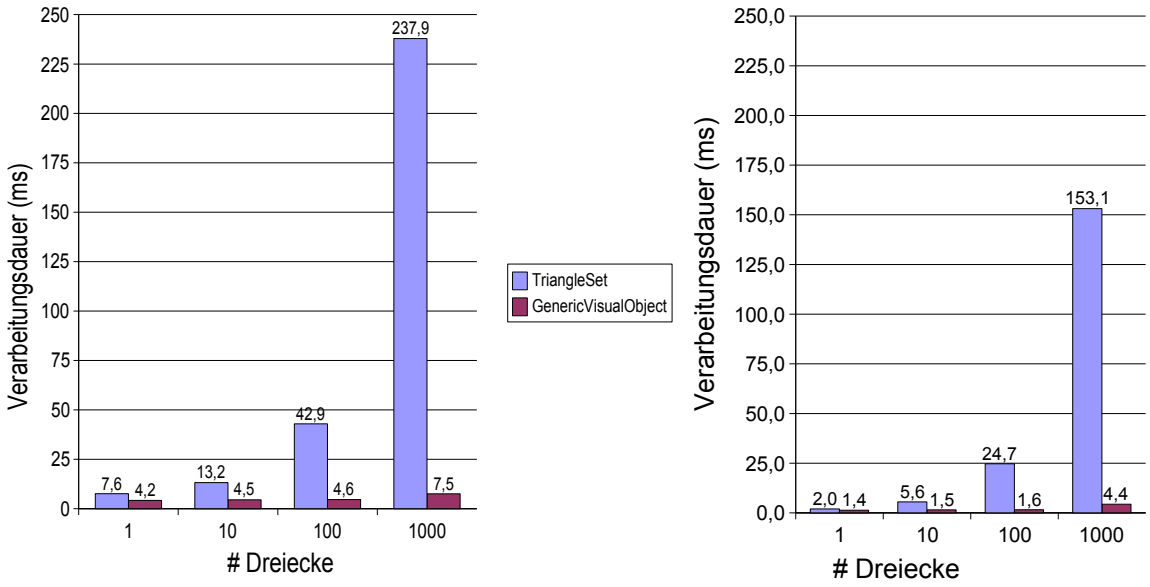
Abbildung 5.15.: Instanzierungsdauer in Abhängigkeit zur Anzahl der Dreiecke

Abbildung 5.15 zeigt die Instanzierungsdauer zweier *VisualObject*s. Es wurden Dreiecksnetze mit unterschiedlicher Anzahl an Dreiecken in verschiedenen Modellierungssprachen definiert. Das *GenericVisualObject* enthält die Defininition eines Dreiecksnetzes in der Modellierungssprache X3D. Das*TriangleSet*-Objekt stellt die in dieser Arbeit entwickelte Möglichkeit dar, Dreiecksnetze zu modellieren. Wie zu erkennen, steigt die Instanzierungsdauer für beide Objekte mit der Anzahl der Dreiecke etwas an. Dies war auch nicht anders zu erwarten. Auffällig jedoch ist, dass das die Instanzierung eines *GenericVisualObject* unabhängig von der Anzahl der Dreiecke über zwei Sekunden benötigt. Der Löwenanteil dieser Zeit wird von der Komponente verursacht, die die X3D-Modelle in Objekte der Java3D-API übersetzt.[6] Weitere Zeit benötigt die Dekodierung der mit Hilfe des Base64-Algorithmus kodierten Modellinformationen. Wegen der hohen Instanzierungsdauer der *GenericVisualObject*s wird hier empfohlen, die Modellinformation nur einmal zu instanzieren und anschließend mit

[6] Als Modell-Parser für X3D wurde der Xj3D-Lader [Con08] des Web3D-Konsortiums verwendet.

Hilfe von *UpdateCommand*s auf das entsprechende Objekt einzuwirken. Das erste Szenario in Kapitel 6 verfolgt diesen Ansatz.

5.2.4. Serialisierung und Deserialisierung



(a) Serialisierung **(b)** Deserialisierung

Abbildung 5.16.: Dauer der Serialisisierung und Deserialisierung

Aus der Dauer für die Instanzierung eines *TriangleSet*-Objektes (siehe Abbildung 5.15) kann man ableiten, dass es sich nicht wirklich lohnt, für *TriangleSet*-Objekte den Aufwand über *UpdateCommands* zu treiben und statt dessen immer wieder die volle Modellinformation zu übertragen. Dies stellt sich jedoch als Trugschluss heraus. Abbildungen 5.16a und 5.16b identifizieren einen weiteren Flaschenhals, der sich bei der Serialisierung bzw. Deserialisierung ergibt. Für die Serialisierung und die Deserialisierung wurden das `ResultSetPlatformSink` und die `ResultSetPlatformSource` verwendet, welchen aber jeweils ein `ResultSet`-Objekt bzw. eine als Byte-Array serialisierte AWML-Datei übergeben wurde, das bzw. die nur ein Objekt enthielt. Die Serialisierungsdauer bzw. die Deserialisierungsdauer der TriangleSet-Objekts vergrößert sich mit der Anzahl der Dreiecke. Das Dreiecksnetz welches mit Hilfe des *GenericVisualObject* definiert wurde, bleibt relativ konstant bei einer niedrigen Verarbeitungszeit und überschreitet für beide Arbeitsschritte nicht 13 ms. Dies ist anhand der Struktur eines *GenericVisualObject*s zu erklären. In Beispiel 5.1 ist exemplarisch ein solches Objekt abgebildet, welches ein Dreiecksnetz in X3D definiert, das anschließend mit dem Base64-Algorithmus kodiert wurde.

Wie zu sehen, ist die gesamte Modellinformation eine für uns unleserliche Zeichenkette. Für den Deserialisier oder Serialisier ist dies lediglich eine Zeichenkette deren Inhalt nicht

interessiert. Ob die Zeichenkette nun 1 KByte oder 100 KByte lang ist, macht dann keinen großen Unterschied mehr. Die Zeichenkette wird gelesen oder geschrieben, so schnell es die unterliegende Hardware zulässt. Das *TriangleSet*-Objekt hingegen definiert die einzelnen Punkte der Dreiecke mit Hilfe von Attributen und Attributteilen. Damit liegt die Modellinformation in einer Struktur vor, die von dem Serialisierer bzw. Deserialisierer interpretiert werden muss. Zudem findet bei dem Prozess die Umwandlung der Zeichenketten in die numerischen Datentypen bzw. der numerischen Typen in Zeichenketten statt. Bei der Verwendung eines *GenericVisualObject*s wird diese Umwandlung nicht beim Serialisieren oder Deserialisieren durchgeführt, sondern erst bei der Instanzierung der Objekte, also bei der Ankunft an der Rendering-Komponente.


```
<awml:nexusobject>
       <nsas:type>
              <nsas:value>ecs:GenericVisualObject</nsas:value>
       </nsas:type>
       ...
       <eas:model>
              <nsas:value>

                     <eas:encoding>BASE64</eas:encoding>
                     <eas:contentType>model/x3d+xml</eas:contentType>
                     <eas:data>
PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz4NCjwhRE9DVFlQRSBYM0QgUFVC
TElDICJJU08vL1dlYjNELy9EVEQgWDNEIDMuMC8vRU4iDQogICJodHRw0i8vd3d3LndlYjNkLm9y
Zy9zcGVjaWZpY2F0aW9ucy94M2QtMy4wLmR0ZCI+DQo8WDNEIHByb2ZpbGU9IkltbWVyc2l2ZSIg
Pg0KCTxoZWFkPg0KCQk8bWV0YSBjb250ZW50PSJTaW1wbGUgWDNEIGV4YW1wbGUiIG5hbWU9ImRl
c2NyaXB0aW9uIi8+DQoJPC9oZWFkPg0KCTxTY2VuZT4NCgkJPFNoYXBlPg0KCQkJPEFwcGVhcmFu
Y2U+DQoJCQk8TWF0ZXJpYWwgZGlmZnVzZUNvbG9yPScwIDAgMScvPg0KCQkJPC9BcHBlYXJhbmNl
PgOKCQkJPEluZGV4ZWRUcmlhbmdsZVNldCBpbmRleD0nMCAxIDInIHNvbGlkPSdmYWxzZScgY2N3
PSd0cnVlJyBjb250YWluZXJGaWVsZD0nZ2VvbWV0cnknPg0KCQkJCTxDb29yZGluYXRlIERFRj0n
UG9pbnRzJyBwb2ludD0nMCAwIDAgMC41IDAgMCAwLjUgMC41IDAnLz4NCgkJCTwvSW5kZXhlZFRy
aWFuZ2xlU2V0PgOKCQk8L1NoYXBlPiANCgk8L1NjZW5lPgOKPC9YM0Q+
                     </eas:data>
              </nsas:value>
       </eas:model>
</awml:nexusobject>
```


Beispiel 5.1: GenericVisualObject im XML-Format

5.2.5. Verarbeitungszeit von Kommandos

Abbildung 5.17 zeigt die durchschnittliche Verarbeitungszeit verschiedener Arten von Kommandos. Als *UpdateCommand* wurde ein Kommando ausgewählt, das eine Koordinate eines Dreiecksnetzes verschiebt. Dies stellt das komplizierteste aller Update-Kommandos dar, da nach jedem Verschieben einer Koordinate eines Dreiecksnetzes dessen Normalen neu berechnet werden müssen, um eine korrekte Darstellung in Gegenwart von Licht zu ermöglichen. Auf Grund dessen, dauert die Verarbeitung des *UpdateCommand*-Objektes am längsten. Insgesamt anzumerken ist, dass die Verarbeitung der Kommandos an der Rendering-Komponente im Vergleich zu der Instanzierung der *VisualObject*s recht schnell vonstatten geht, weswegen

es auch sinnvoll ist, die Instanzierung der Objekte auf ein Minimum zu reduzieren und wenn möglich mit Update-Kommandos zu arbeiten.

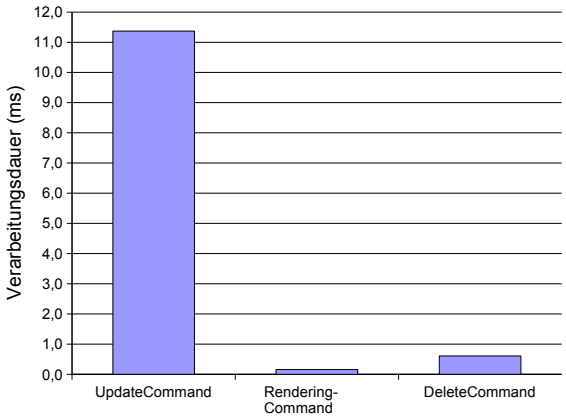


Abbildung 5.17.: Verarbeitungszeit verschiedener *Command*-Objekte

5.2.6. Benötigte Bandbreite der Netzwerkverbindung

Große Datenmengen werden bei der Übermittlung des JPEG-Videostrom übertragen. Hier lassen sich rechnerisch gewisse Mindestanforderungen an die Übertragungsraten der unterliegenden Netzwerktechnologie festlegen. Dabei wird von den verschiedenen Bildraten der JPEG-Komprimierung bei verschiedenen Auflösungen ausgegangen (siehe Abbildung 5.14). Die einzige Ausnahme stellt hier die Bildrate der ersten Auflösung dar. Hier liegt die JPEG-Komprimierungsrate über der Bildrate des Rendering-Prozesses. Da die Bilder nicht schneller komprimiert als produziert werden können, wurde hier die Bildrate des Rendering-Prozesses übernommen. Die durchschnittliche Größe der komprimierten JPEG-Bilder des rotierendes Würfels bei verschiedenen Auflösungen wurde durch Messung ermittelt.

$$B = R \cdot (S + H)$$

Die benötigte Bandbreite B ist das Produkt der Bildrate R mit der Größe des Bildes S zuzüglich eines konstanten Header H von 20 Byte. In Tabelle 5.2 sind die benötigten Bandbreiten aufgelistet. Zwar sind diese Ergebnisse abhängig von der hier verwendeten Hardware, jedoch lässt sich die Erkenntnis ziehen, dass die Bandbreite der verwendeten Netzwerk-Technologie in Zukunft eher keinen Flaschenhals darstellen wird. Selbst unter der Annahme, dass die Bilder schneller produziert werden können und so zum Beispiel Bildraten von 20 fps bei einer Auflösung von 1280×800 Pixel erzielt werden, so würde die benötigte Übertragungsrate immer noch „nur“ ca. 800 KByte/s betragen, die sich zum Beispiel mit einem 10 MBit-Ethernet realisieren ließe.

Auflösung	Dateigröße (KByte)	Bildrate (fps)	Übertragungsrate (KByte/s)
256 x 256	4	60,2	242
512 x 512	8	32,1	257
800 x 600	16	17,8	285
1024 x 758	28	10,7	299
1280 x 800	40	8,2	328

Tabelle 5.2.: Benötigte Übertragungsraten bei verschiedenen Auflösungen und Bildraten

5.2.7. Bildrate des Empfängers

Viel brisanter als die Frage nach der Übertragungsgeschwindigkeit, ist die Frage, ob ein Client, der möglicherweise nur beschränkte CPU-Kapazitäten inne hat, die ankommenden Daten schnell genug wiedergeben kann. Dies wird im Folgenden untersucht.

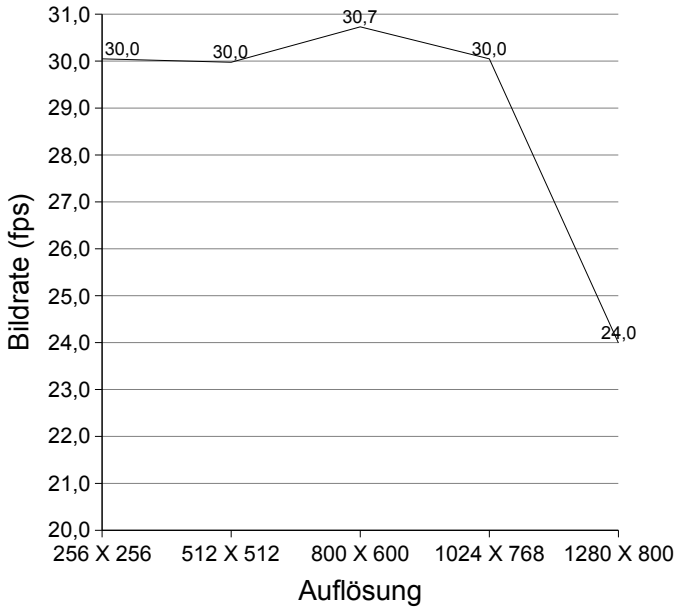


Abbildung 5.18.: Wiedergaberate beim Abspielen

Abbildung 5.18 zeigt die durchschnittliche Bildrate des `VideoViewerSinks` bei verschiedenen Auflösungen. Auffällig ist, dass die Bildrate bei den geringen Auflösungen relativ konstant bei 30 fps verweilt. Dies ist wohl mit den Eigenschaften der verwendeten Player-Software zu erklären. Es wurde die Player-Software von JMF verwendet. Es hat den Anschein, dass diese Komponente Videos grundsätzlich nicht schneller als in einer Rate von 30 fps wiedergibt.

Unabhängig davon wird jedoch sichtbar, dass bei einer Auflösung von 1280×800 Pixeln, die Bildrate deutlich abfällt. Daraus kann man schließen, dass hier die maximal mögliche Auflösung mit der hier verwendeten Hardware-Umgebung A allmählich erreicht wird. Dem

JMF-Player werden die Bilder als in Byte-Arrays kodierte JPEGs übergeben. Was intern mit diesen Bildern passiert, ist unbekannt. Da sie aber im JPEG-Format übergeben werden, muss an irgendeiner Stelle eine JPEG-Dekodierung stattfinden. Es liegt die Vermutung nahe, dass für den Abfall der Bildrate die Dekodierung der JPEG-Bilder verantwortlich ist.

Schlussendlich kann man erkennen, dass es auch beim Abspielen eine maximal mögliche Grenze der Auflösung gibt. Auch wenn diese mit der hier verwendeten Hardware noch nicht ganz erreicht wurde, so ist dies aber wahrscheinlich, wenn Hardware verwendet wird, die eine leistungsschwache CPU besitzt.

5.2.8. Latenz der Interaktion

Der Betrachter einer Visualisierung möchte gegebenfalls in dem visualisierten Szenario interaktiv navigieren. Besonders aufschlussreich sind nun Messungen, die die maximale zu erwartende Zeit einer Navigationsanfrage, bis zu ihrer sichtbaren Umsetzung eingrenzen. Es gibt drei Möglichkeiten eine Navigationsanfrage durchzuführen:

1. Der Client formuliert eine Update-Anfrage und schickt diese an den Föderations-Manager, der die benötigten Parameteränderungen extrahiert, und diese den Operatoren mitteilt, die auf den von ihm verwalteten Knoten ausgeführt werden.
2. Der Client benutzt eine direkte Kontrollleitung zum `RenderOperator` und teilt die Navigationswünsche diesem mit Hilfe der *RenderingCommand*-Kommandos mit.
3. Der Client führt selbst die Bildgenerierung durch und navigiert mit Hilfe des dort gestarteten Tastatur-Navigators.

Die Latenz der ersten Möglichkeit zu messen, ist im Zuge dieser Arbeit nicht realisierbar. Bis jetzt gibt es weder eine visuelle Anfragesprache noch einen Föderations-Manager, der die Parameteränderungen den Knoten mitteilen könnte. Wurden Anfragen optimiert, so dass für verschiedene Clients, gleiche Teile des physischen Operatorbaums verwendet werden, muss zusätzlich noch geprüft werden, ob die Parameteränderung der Update-Anfrage, die Anfragen anderer Clients semantisch beeinflussen würde. Deswegen wird hier davon ausgegangen, ohne sich auf konkrete Messergebnisse stützen zu können, dass diese Variante die mit der größten zu erwartenden Latenz darstellt.

Die letzte Möglichkeit wurde im Zuge dieser Arbeit realisiert. Jedoch stellt sich eine exakte Messung als schwierig heraus. Der Bildgenerierungs-Prozess des Java3D-Subsystems läuft unabhängig von dem üblichen Programmfluss. Die exakte Zeit, bis das Drücken einer Taste zu einer sichtbaren Veränderung auf dem Bildschirm führt, ist unbekannt, da keine genaue Kenntnis über den Zustand des Java3D-Subsystem, zu einem bestimmten Zeitpunkt vorliegt. Für die menschliche Perzeption geschieht die Umsetzung jedoch sofortig, was auf eine Reaktionszeit von weniger als $0,1$ Sekunden rückschließen lässt.

Nur für die zweite Variante ist es lohnenswert, Messungen anzustellen. Vorab sollte geklärt werden, an welchen Stellen Latenzen bei der Verwendung einer direkten Kontrollleitung entstehen können.

1. Der Client erstellt nach dem Empfang eines Ereignisses, welches den Navigationswunsch des Betrachters enthält, ein *RenderingCommand*-Objekt. Die Latenz bei der Erstellung des *RenderingCommand*s als eine Instanz von `GenericObject` betrug weit weniger als 1 ms und kann daher vernachlässigt werden.
2. Das *RenderingCommand* wird über das `GenericObjectPlatformSink` serialisiert und an das gegenüberliegende `GenericObjectPlatformSource` übertragen und dort deserialisiert. Die Dateigröße eines *RenderingCommand*s entspricht ungefähr der eines *TriangleSet*-Objekts, das ein Dreieck enthält. Die Latenzen, die bei der Deserialisierung und Serialisierung eines TriangleSet-Objektes entstehen, wurden hier schon untersucht (siehe Abbildung 5.16) und betragen insgesamt weniger als 10 ms. Die Übertragungsdauer des unterliegenden Netzwerks ist unbekannt und sicherlich abhängig von der Netzwerktechnologie und der verwendeten Verbindung. Findet eine Übertragung über das Internet statt und sind die beiden Rechner physisch weit voneinander entfernt, so muss mit einer hohen Latenz gerechnet werden.
3. Anschließend wird das *RenderingCommand* vom `RenderOperator` verarbeitet. Die Änderung hat sich nach dem nächsten Bildgenerierungs-Zyklus in den generierten Bildern niedergeschlagen. Die Verarbeitungszeit eines *RenderingCommand*s ist in Abbildung 5.17 aufgetragen und liegt weit unter 1 ms, weshalb diese Zeit auch vernachlässigt werden kann. Der nächste Bildgenerierungs-Zyklus findet bei einer Rate von 40 fps spätestens nach 25 ms statt.
4. Die generierten Bilder müssen nun in JPEG-Bilder komprimiert werden. Die Latenz hierbei berechnet sich aus der inversen Bildrate der verwendeten Auflösung aus Abbildung 5.14. Bei einer Auflösung von 512×512 Pixel und einer Bildrate von 32 fps beträgt die Latenz zum Beispiel ca. 31 ms.
5. Die JPEG-Bilder werden nun wieder über ein Netzwerk an den Client übertragen. Die Latenz, die hierbei entsteht, ist wiederum unbekannt.
6. Beim Client werden die JPEG-Bilder von der Player-Software dekodiert und angezeigt. Bei einer Bildrate von 24 fps entsteht hier eine Latenz von ca. 42 ms.

Die oben beschriebenen Latenzen treffen nur zu, wenn das Senden und Empfangen von Objekten zwischen den Operatoren innerhalb eines Knotens keine Zeit benötigt. Dies ist allerdings nur dann nahezu verlustfrei, wenn die Warteschlangen zwischen den Operatoren leer sind, bzw. maximal ein Element enthalten. Im Allgemeinen sollte dieser Zustand angestrebt werden, da sonst auf lange Sicht hin Datenverlust entstehen kann. Deswegen wird im Folgenden davon ausgegangen, dass die Auflösung so gering gewählt wurde, dass sich an keiner Stelle in der beschriebenen Verarbeitungskette die Daten aufstauen.

Folglich bleibt als unbekannter Anteil der Interaktions-Latenz die Übertragungsdauer über das Netzwerk. Dementsprechend ist es lohnenswert, genau diese zu messen. Hierfür wurde eine Konfiguration gewählt, die prinzipiell dem Senden einer Ping-Nachricht entspricht (siehe Abbildung 5.19).

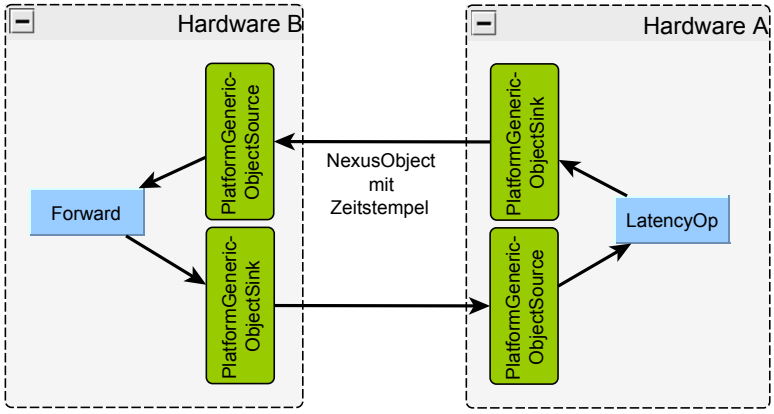


Abbildung 5.19.: Aufteilung zur Messung der Latenz

Hierfür wird dem `LatencyOp`-Operator periodisch ein *NexusObject* gesendet, das einen Zeitstempel enthält. Dieses wird von dem `PlatformGenericObjectSink`- und `Source` serialisiert, übermittelt und deserialisiert. Der in [Sar08] schon verwendete `Forward`-Operator leitet das Objekt direkt an den Knoten der Hardware-Umgebung A zurück. Der `LatencyOp`-Operator misst dann die Zeitdifferenz.

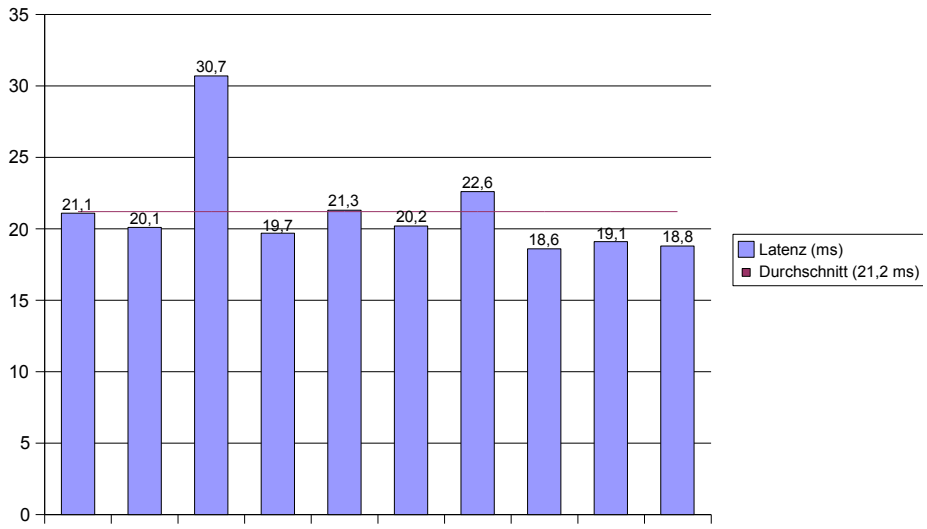


Abbildung 5.20.: Latenz

Wie in Abbildung 5.20 ersichtlich, liegt die durchschnittliche Latenz von 10 Messungen bei ca. 21 Millisekunden. Hierbei ist allerdings die Serialisierungsdauer und Deserialisierungsdauer des *NexusObject*s enthalten. Das übermittelte *NexusObject* lässt sich von der Größe und Komplexität eher mit dem *GenericVisualObject* vergleichen, das ein Dreieck enthält (siehe Abbildung 5.16). Da insgesamt zweimal serialisiert und deserialisiert werden muss, ergibt sich für die aktuelle Übertragungsdauer über das Netzwerk eine Latenz von weniger als

10 Millisekunden. Getestet wurde über ein WLAN-Netzwerk, das dem Standard von IEEE 802.11g [IEE07] entspricht.

Zusammenfassend lässt sich also feststellen, dass bei geringen Auflösungen und nicht allzu großer Latenz im Netzwerk, bei der Interaktion Latenzen erzielt werden können, die klar unter einer Sekunde liegen. Ist kaum Netzwerk-Latenz vorhanden, so kann die Grenze von 0,1 Sekunden erreicht bzw. sogar unterschritten werden. Netzwerk-Latenz entsteht typischerweise immer dann, wenn die Netzwerk-Pakete zwischen verschiedenen Netzwerken vermittelt werden müssen, oder wenn zwischen den Knoten eine große geographische Distanz besteht und sich die Signallaufzeiten bemerkbar machen. Befinden sich die Knoten jedoch im selben Subnetz, was in der Regel auch geographische Nähe impliziert, so ist kaum Latenz zu erwarten.

Kapitel 6

Visualisierungs-Beispiele

In diesem Kapitel sollen beispielhaft einige Möglichkeiten zur Aufteilung der Komponenten in einem verteilten Umfeld aufgezeigt werden. Die Grundlage hierfür bilden zwei Visualisierungs-Beispiele, die im Folgenden vorgestellt werden. Es werden dabei nicht ausschließlich die Standard-Komponenten verwendet, die in Kapitel 5 detailliert beschrieben wurden. Es wurde zusätzlich an einigen Stellen Funktionalität benötigt, die spezifisch für das Anwendungsfeld ist und deswegen für andere Visualisierungs-Beispiele mit hoher Wahrscheinlichkeit nicht verwendet werden kann. Deswegen wurde darauf verzichtet, die spezifischen Komponenten auf Implementierungs-Ebene zu beschreiben. Hier werden lediglich die wesentlichen Konzepte, die hinter einer solchen spezifischen Komponente stehen, erläutert.

6.1. Visualisierung von mobilen Objekten

Die Visualisierung von mobilen Objekten kann zum Beispiel für Untersuchungen des maximalen Verkehrsdurchsatzes einer bestimmten Straße oder Kreuzung genutzt werden. Eine Simulation könnte entsprechend gewissen Parametern, bestimmte Verkehrslasten erzeugen. Die Visualisierung kann dann verwendet werden, um visuell festzustellen, an welchen Stellen sich Autos aufstauen, oder ob andere Verkehrswege blockiert werden. Wenn gewisse Problemzonen visuell entdeckt wurden, könnte man durch Navigieren entlang der Problemzonen das genaue Ausmaß erkennen. Eine realistische, dreidimensionale Visualisierung könnte hier den Wiedererkennungswert einer Problemzone in einer Stadtregion steigern.

6.1.1. Problem

Realistische, dreidimensionale Visualisierungen beinhalten komplexe, dreidimensionale Modelle, die auch Texturen enthalten können. Es liegt auf der Hand, dass versucht werden muss, diese Modelle nur so oft wie nötig zwischen den beteiligten Knoten des Visualisierungsnetzes auszutauschen. Wird der Verkehr durch realistische Modelle von Autos

dargestellt, so sollte darauf geachtet werden, dass der Komponente, welche die Bildgenerierung betreibt, nur einmal die Daten des dreidimensionalen Modells eines Autos übermittelt werden. Werden die Modelle mit Hilfe des *GenericVisualObject*s übertragen, so ist wie in Kapitel 5.2 festgestellt, die hohe Instanzierungsdauer das Haupthindernis einer fortwährenden Übermittlung der Modellinformationen.

6.1.2. Lösung

Alle statischen Objekte des visualisierten, dreidimensionalen Szenarios, wie Straßen, Gebäude, oder ähnliches werden dem RenderOperator zu Beginn einmal mitgeteilt. Nun muss darauf geachtet werden, dass auch die Modelldaten der mobilen Objekte nur einmal übermittelt werden.

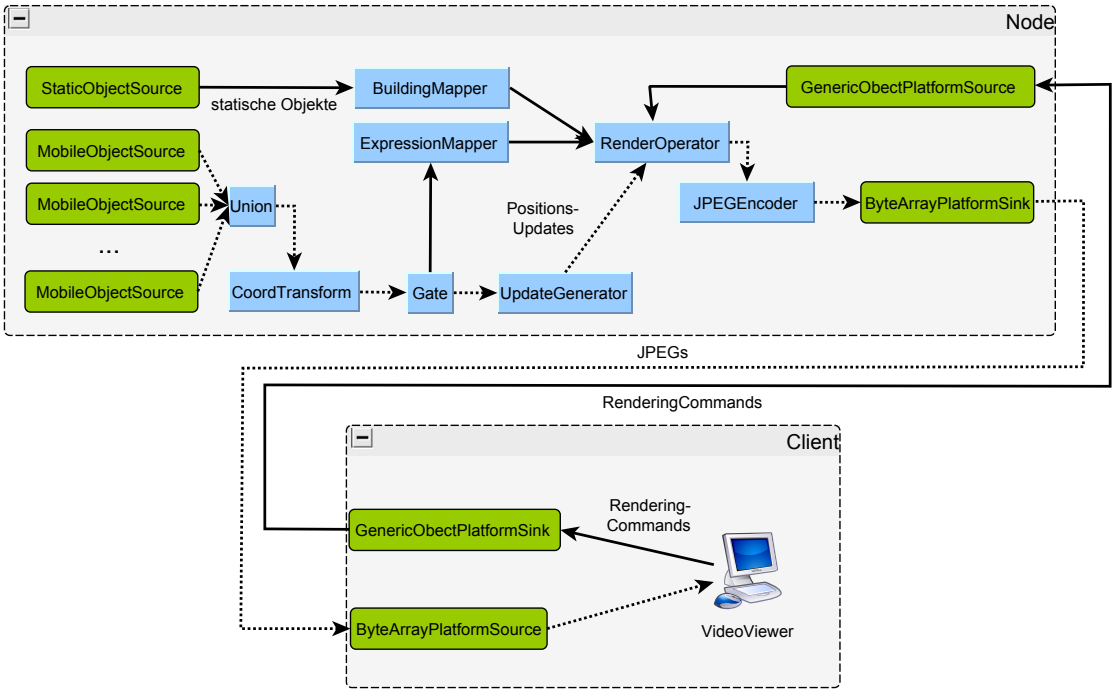


Abbildung 6.1.: Operatorgraph zur Visualisierung von mobilen Objekten

Abbildung 6.1 zeigt eine mögliche Aufteilung, um die Visualisierung der mobilen Objekte vorzunehmen. Statische Objekte wie Häuser oder Straßen werden von der `StaticObjectSource` erzeugt. Die `StaticObjectSource` extrahiert die statischen Objek-

tes aus einer AWML-Datei und schickt diese als `ResultSet`-Objekt[1] an seinen Nachfolger, den `ExpressionMapper`. Der `BuildingMapper`-Operator funktioniert vom Prinzip her ähnlich wie der `ExpressionMapper`-Operator. Der Operator wird initialisiert mit einem *MapTo*-Objekt (siehe Kapitel 5.1.3). Dieses dient als Schablone, die die statischen Attribute der visuellen Darstellung enthält. Die zweidimensionale Darstellung der Gebäude wird unter Angabe einer Höhe auf eine dreidimensionale Darstellung abgebildet. Er erwartet an seinem Eingang eine `ResultSet`-Instanz. Objekte mit dem Typ *Building* werden auf die dreidimensionale Darstellung abgebildet und mit den statischen Attributen des `MapTo`-Objektes versehen. Anschließend werden die so erstellten *VisualObject*s der Gebäude in eine `ResultSet`-Instanz eingefügt und an den nachfolgenden `RenderOperator` gesendet, um sie dort darzustellen.

Für die hochdynamische Positionsinformation der mobilen Objekte wurde eine spezielle Datenquelle entwickelt, die `MobileObjectSource`. Mit Hilfe des Simulationswerkzeuges CanuMobiSim [SHB+03] wurden Trajektorien von Objekten erstellt, die sich auf den Straßen der Innenstadt Stuttgarts bewegen. Diese Trajektorien werden nun von der `MobileObjectSource` verwendet, um unter Angabe eines Sende-Intervalls die Positionsinformationen der mobilen Objekte in bestimmten Abständen an den nachfolgenden Operator, den `Union`-Operator, zu senden. Dieser vereinigt die eingehenden Datenströme der Eingänge in einen einzigen Datenstrom, der die Positionsinformationen aller mobilen Objekte enthält. Die Positionen dieses Datenstroms werden mit Hilfe des `CoordTransform`-Operators in ein kartesisches Koordinatensystem transformiert und anschließend dem `Gate`-Operator überreicht. Dieser sorgt nun dafür, dass der erste Wert eines mobilen Objektes, welches durch seine NOL identifiziert wird, an die `ExpressionMapper`-Instanz gesendet wird und alle darauf folgenden Objekte an den `UpdateGenerator` weitergeleitet werden. Der erste Positionswert eines Objekts wird mit Hilfe der `ExpressionMapper`-Instanz unter Angabe von Abbildungsvorschriften auf ein visuelles Modell einer externen Modellierungssprache mit Hilfe des *GenericVisualObject* abgebildet, wie zum Beispiel das Modell eines Automobils. Das so gebildete *VisualObject* wird nun der `RenderOperator`-Instanz übergeben und sofort angezeigt. Um zu vermeiden, dass die Modellinformationen der mobilen Objekte, welche prinzipiell sehr groß sein können, immer wieder gesendet werden und von der `RenderOperator`-Instanz erneut gelesen werden müssen, leitet der `Gate`-Operator alle folgenden Positionsinformationen über den `UpdateGenerator` zum `RenderOperator`. Die `UpdateGenerator`-Instanz generiert nun *UpdateCommand*-Objekte, welche den `RenderOperator` dazu veranlassen, das Positionsattribut der angezeigten Objekte zu ändern. Letztlich werden die Modelldaten also nur einmal übertragen, während die Positionen der Objekte über spezielle *Command*-Objekte aktualisiert werden.

Die vom `RenderOperator` erstellten Bilder werden in JPEGs umgewandelt und mit Hilfe des `ByteArrayPlatformSinks` an den Client gesendet. Diese werden mit Hilfe des `VideoViewer`-Operators dargestellt. Bei Navigationsoperationen werden *RenderingCommand*-Objekte erstellt und an die `RenderOperator`-Instanz gesendet.

Abbildung 6.2 zeigt die Visualisierung von den Positionsinformationen zweier mobiler Objekte, die für die Visualisierung auf ein Modell eines Automobils abgebildet wurden. Die

[1] Ein `ResultSet`-Objekt ist die deserialisierte Form einer AWML-Datei und besteht aus einer Menge von `GenericObject`-Instanzen.

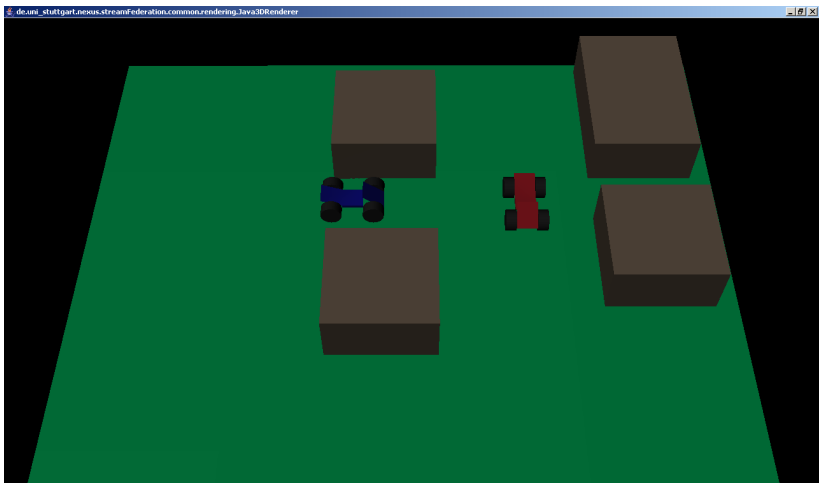


Abbildung 6.2.: Visualisierung von mobilen Objekten

Modelle der Automobile wurde in X3D modelliert. Die zweidimensionalen Grundrisse der Gebäude wurden mit Hilfe des `BuildingMapper`-Operators mit einer Höhe versehen.

6.2. Höhenfeld

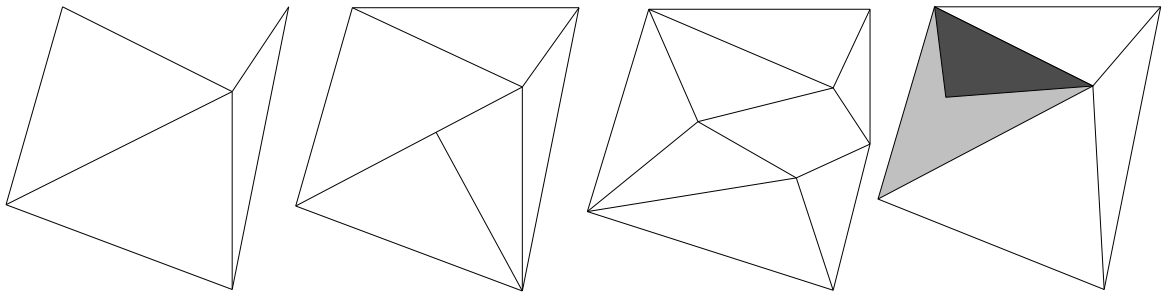
Im Gegensatz zur Visualisierung von mobilen Objekten, die ihre Position ständig ändern, basiert die Visualisierung mit Hilfe eines Höhenfeldes auf skalaren Werten, die fest mit einer zweidimensionalen Position verbunden sind. Dies könnten zum Beispiel die Signalstärken eines Mobilfunknetzes oder gemessene Temperaturen an bestimmten Positionen in einem Gebiet sein. Diese Daten sind zunächst einmal verstreut, das heißt, sie besitzen keinerlei Topologie-Information. In einem ersten Schritt muss eine Topologie hergestellt werden. Aus den verstreuten Daten wird mit Hilfe eines Triangulations-Algorithmus ein Dreiecksnetz erzeugt. Das Dreiecksnetz ist zunächst noch zweidimensional, was es ermöglicht, den zu visualisierenden Skalar auf die dritte Koordinate, die Höhe, abzubilden. Der Skalar kann viele Bedeutungen haben. Im Folgenden wird exemplarisch von Temperaturwerten ausgegangen, die an verschiedenen Positionen gemessen wurden.

6.2.1. Triangulationsverfahren

Es sind zahlreiche Triangulationsverfahren bekannt. An dieser Stelle soll jedoch nur das auch in dieser Arbeit verwendete erwähnt werden. Dieses wurde wegen seiner einfachen Umsetzungsmöglichkeit gewählt, nicht auf Grund der Laufzeit des Algorithmus. Der folgende Algorithmus besitzen im schlechtesten Fall eine quadratische Laufzeit.

Gültiges Dreicksnetz

Bevor Triangulationsverfahren beschrieben werden, sollte zunächst definiert werden, was man unter einem gültigen Dreiecksnetz versteht. Alle Triangulationsverfahren sollten eine Menge unabhängiger Punkte in ein gültiges Dreiecksnetz abbilden. Vereinfacht gesagt, darf ein gültiges Dreiecksnetz keine Einbuchtungen, Löcher, überlappende Dreiecke oder T-Knoten enthalten (siehe Abbildung 6.3).



(a) Einbuchtung **(b)** T-Knoten **(c)** Loch **(d)** Überlappung

Abbildung 6.3.: Invalide Dreiecksnetze

Damit müssen gültige (zweidimensionale) Dreiecksnetze folgende Kriterien erfüllen:

1. Der Schnitt zweier Dreiecke ist entweder leer, ein gemeinsamer Eckpunkt, eine gemeinsame Kante oder die gemeinsame Dreiecksfläche.
2. Alle Kanten, die nicht an ein anderes Dreieck angrenzen, sind auch Kanten der konvexen Hülle der Eckpunkte aller Dreiecke.

Dabei ist die *konvexe Hülle* einer Punktmenge V das kleinste konvexe Polygon, für das sich jeder Punkt aus V entweder auf dem Rand des Polygons, oder in dessen Inneren befindet (siehe hierzu auch [CLRS04]).

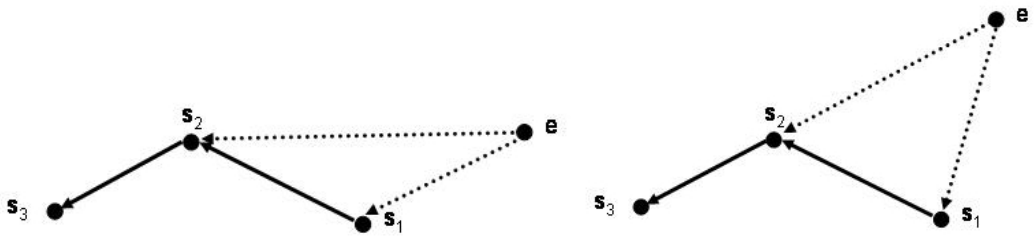
PlaneSweep

Mit dem *PlaneSweep*-Triangulationsverfahren nach [For86] kann ein gültiges Dreicksnetz gebildet werden. Hierzu werden die Punkte der Eingabemenge nach der X-Koordinate von links nach rechts sortiert. Reicht dies nicht als Kriterium aus, so wird zusätzlich nach der Y-Koordinate sortiert. Dann werden die Punkte aus der sortierten Menge sukzessive betrachtet. Bildlich kann man sich eine vertikale Linie vorstellen (*SweepLine*), die sich von links nach rechts durch die Menge der Punkte bewegt. Für alle Punkte, die links der Linie liegen, wurden die Dreiecke schon gebildet, die Punkte rechts davon müssen noch in Betracht gezogen werden. Der nächste zu bearbeitende Punkt ist der Punkt, der der Linie von rechts am nächsten liegt. Falls hier mehrere Punkte in Betracht kommen, diese also den gleichen Abstand zur vertikalen Linie besitzen, so wird der unterste gewählt. Der letzte Punkt, der

dem Dreiecksnetz hinzugefügt wurde, ist nun unser Ausgangspunkt. Von ihm bewegen wir uns entgegen dem Uhrzeigersinn, bis der oberere Tangentenpunkt der konvexen Hülle des Dreiecksnetzes erreicht ist. Die dabei besuchten Punkte der konvexen Hülle sind Eckpunkte der zu bildenden Dreiecke und bilden mit dem nächsten zu bearbeitenden Punkt eine Kante der Dreiecke, welche als nächstes dem Dreiecksnetz hinzugefügt werden. Analog verfährt man mit den Punkten, die im Uhrzeigersinn auf der konvexen Hülle liegen.

Um festzustellen, ob der obere bzw. untere Tangentenpunkt mit der konvexen Hülle erreicht ist, kann nach [CLRS04] mit Hilfe spezieller Eigenschaften des Kreuzprodukts zweier Vektoren entschieden werden. Für zwei Vektoren v_1 und v_2 im $\mathbb{R}^2$ ist das Kreuzprodukt als die Determinante der Matrix (v_1, v_2) definiert. Also gilt: $v_1 \times v_2 = x_1 y_2 - x_2 y_1$.

Der Wert des Kreuzprodukts ist der Flächeninhalt des Parallelogramms, das die beiden Vektoren aufspannen. Dieser ist positiv, wenn sich v_2 gegenüber v_1 bezüglich des Ursprungs entgegen dem Uhrzeigersinn befindet, negativ wenn er sich im Uhrzeigersinn befindet. Das Kreuzprodukt hat den Wert 0, wenn die Vektoren kollinear sind.



(a) Rechts - Links **(b)** Rechts - Kollinear

Abbildung 6.4.: Oberer Tangentenpunkt

Wie man das Kreuzprodukt für die Tangentpunktbedingung ausnutzen kann, lässt sich in Abbildung 6.4 erkennen. Hier wendet sich für den oberen Tangentenpunkt die Strecke (s_1, s_2) gegenüber der Strecke (e, s_1) nach rechts, während sich die Strecke (s_2, s_3) gegenüber der Strecke (e, s_2) nach links wendet. Einen Spezialfall stellt die Kollinearität zweier Vektoren dar. Bezieht man diesen Spezialfall mit ein, so muss sich das zweite Streckenpaar nicht unbedingt nach links wenden, sondern kann auch kollinear sein.

Einschränkungen

Der oben beschriebene Algorithmus generiert zwar ein gültiges Dreiecksnetz, das auch angezeigt werden kann. Dennoch kann es vorkommen, dass Dreiecke entstehen, die hinsichtlich ihrer Form für die Darstellung nicht ganz optimal sind. Winkel, die im Vergleich zu den anderen Winkeln eines Dreiecks besonders groß oder klein sind, sind in der Regel unerwünscht, da dies bei der Darstellung besonders in Gegenwart von Licht zu unschönen Nebeneffekten wie Grate, oder sichtbaren Kanten kommen kann. Für ein optimales Dreiecksnetzes hinsichtlich der Winkel wurden in [Del34] gewisse Eigenschaften definiert, die so genannten Delaunay-Eigenschaften, die ein Dreiecksnetz erfüllen muss. Um aus

dem vom Plane-Sweep-Algorithmus erstellten gültigen Dreiecksnetz ein optimales Dreiecksnetz im Sinne der Delaunay-Eigenschaften herzustellen, kann zum Beispiel der Edge-Flip-Algorithmus nach [Law72] verwendet werden. Für diese Arbeit ist ein Erzeugen eines optimalen Dreiecksnetzes jedoch weniger wichtig, da hier nur demonstriert werden soll, wie Daten durch die Triangulation aggregiert werden können, um so auf effizientere Weise dargestellt werden zu können. Weiterhin soll hier aufgezeigt werden, wie durch geschickte Anordnung im Operatorbaum die Anzahl der durchzuführenden Triangulationen minimiert werden kann.

6.2.2. Problem

Wie schon erwähnt, hat der hier verwendete Algorithmus im schlechtesten Fall quadratische Laufzeit. Es sind noch schnellere Algorithmen bekannt, mit einer Komplexität von beispielsweise $\mathcal{O}(n \; log \; n)$. Selbst wenn es einen Algorithmus mit linearer Komplexität gäbe, so wäre die Triangulation zu teuer, wenn man bedenkt, dass in einem realen Szenario sehr viele Sensoren mit hohen Aktualisierungsraten neue Werte produzieren.

Diese teure Operation sollte wenn möglich nicht von der Rendering-Komponente durchgeführt werden. Dieser ist schon beschäftigt mit der vergleichsweise komplexen Aufgabe der Bildgenerierung. Außerdem würde eine Triangulation unabhängiger Datenpunkte einen erhöhten Speicherbedarf nach sich ziehen. Denn für jedes angezeigte *VisualObject* werden von der Rendering-Komponente gewisse Datenstrukturen generiert, die später zum Beispiel eine Positionsänderung oder Farbänderung ermöglichen. Damit müsste die Rendering-Komponente für jeden unabhängigen Datenpunkt Informationen speichern, wie den Identifizierer des Punktes, die aktuelle Transformationsmatrix und Materialeigenschaften, wie Farbe, Reflexionseigenschaften, oder Transparenzeigenschaften.

6.2.3. Lösung

Aus den oben beschriebenen Herausforderungen lassen sich zwei Schlussfolgerungen ziehen:

1. Die Triangulation sollte nicht von dem `RenderOperator` durchgeführt werden, sondern extern mit einem dafür speziell entworfenen Operator.
2. Die unabhängigen Datenpunkte sollten zu einem einzigen Dreiecksnetz-Objekt aggregiert werden, das durch eine einzige NOL identifiziert wird. Damit gibt es den Speicher-Overhead für die Objekt-Eigenschaften nur einmal.

In Abbildung 6.5 ist der Übersicht wegen nur der logische Operatorgraph der Visualisierung eines Höhenfeldes zu sehen. An dessen Ursprung stehen mehrere `ThermometerSource`-Instanzen. Ein `ThermometerSource` generiert *Thermometer*-Objekte. Ein *Thermometer*-Objekt enthält die NOL des Thermometers und einen skalaren Wert, der die Temperatur widerspiegelt. Ein `ThermometerSource` generiert zufällig Werte mit einer Rate, die durch einen Parameter festgelegt wurde. Die Datenströme der verschiedenen Quellen werden mit Hilfe

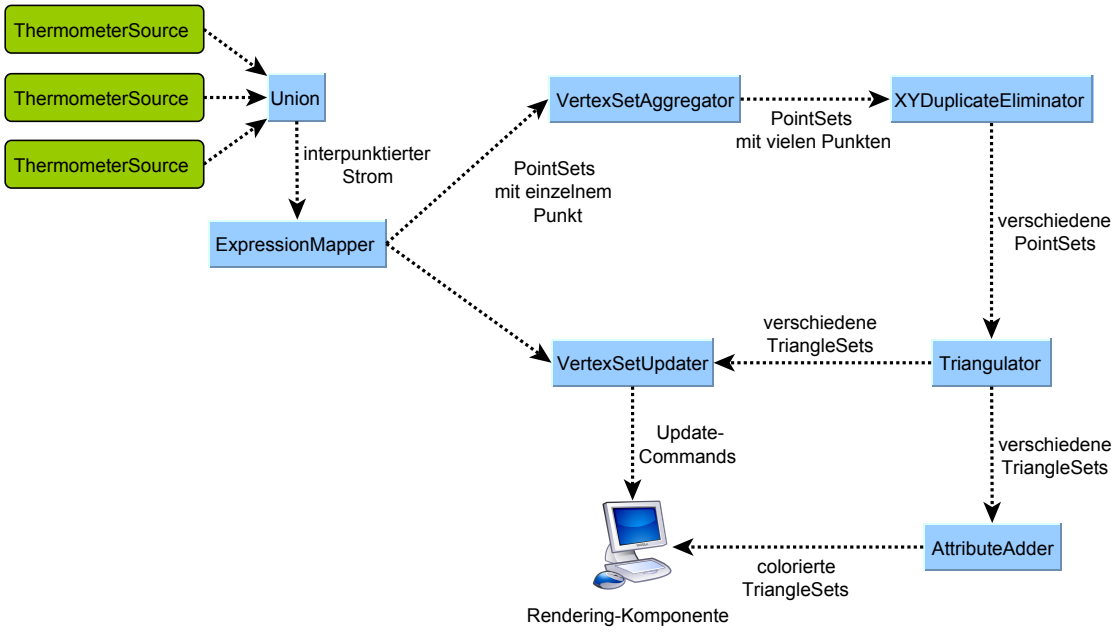


Abbildung 6.5.: Logischer Operatorgraph zur Visualisierung von Höhenfeldern

des `Union`-Operators vereinigt. Ziel ist es nun, die Werte der verschiedenen Datenquellen zu aggregieren. Dies stellt sich als nicht trivial heraus. Denn die Komponente, welche die Aggregation vornimmt, hat nicht das benötigte Wissen, um zu entscheiden, welche Punkte aggregiert werden sollen. Für diese Komponente ist der vereinigte Datenstrom ein potenziell unendlicher Datenstrom von Temperatur-Werten. Es ist nun aber genau dann sinnvoll die Aggregation vorzunehmen, wenn von jeder Datenquelle mindestens ein Wert empfangen wurde. Dieses Wissen hat nur der `Union`-Operator. Er kann einfach feststellen, wann von allen Eingängen ein Wert empfangen wurde und dann den resultierenden Datenstrom an den entsprechenden Stellen markieren. Diese Herangehensweise ähnelt dem Interpunktions-Ansatz (siehe Kapitel 2.3.3). Als Interpunktions-Objekte werden *CacheCommand*- und *FlushCommand*-Objekte verwendet (siehe Kapitel 3.2.5). In Abbildung 6.6 ist die Arbeitsweise des `Union`-Operators zu sehen.

Der so generierte interpunktierte Datenstrom wird an den `ExpressionMapper` weitergeleitet. Dieser ignoriert die *CacheCommand*- und *FlushCommand*-Objekte und leitet sie weiter. Die Weiterleitung unbekannter Kommandos ist die generelle Vorgehensweise der Operatoren. Alle anderen Objekte, in diesem Fall `Thermometer`-Objekte, werden der Abbildung gemäß einer Abbildungsvorschrift unterzogen. In diesem Fall werden die *Thermometer*-Objekte in ein *PointSet*-Objekt abgebildet, welches nur einen Punkt enthält. Die dritte Koordinate des Punktes ist der skalierte Temperaturwert des *Thermometer*-Objektes. Der Ausgangsdatenstrom der `ExpressionMapper`-Instanz wird nun einerseits an den `VertexSetAggregator` und andererseits an den `VertexSetUpdater` weitergeleitet. Der `VertexSetAggregator` aggregiert nun alle Objekte, die sich zwischen einem *CacheCommand*- und *FlushCommand*-Objekt befin-

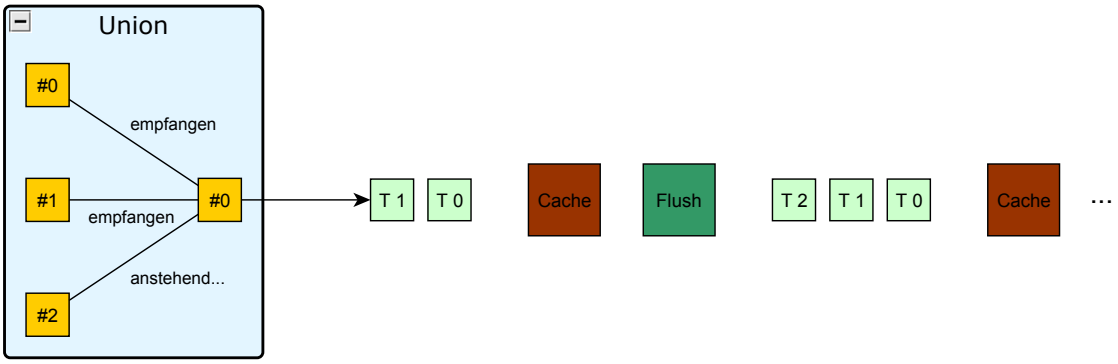


Abbildung 6.6.: Arbeitsweise des `Union`-Operators

den, in ein einziges *PointSet*-Objekt. Das *CacheCommand*- und *FlushCommand*-Objekt werden dabei aus dem resultierenden Datenstrom entfernt. Die *PointSet*-Objekte werden von dem `XYDuplicateEliminator` selektiert. Dieser speichert das zuerst empfangene *PointSet*-Objekt in seinem lokalen Speicher und leitet es weiter. Alle folgenden Objekte werden mit dem PointSet-Objekt im Speicher verglichen. Nur wenn das empfangene PointSet-Objekt sich von dem PointSet-Objekt im Speicher unterscheidet, ersetzt der Operator das PointSet-Objekt im Speicher mit dem neuen Objekt und leitet das neue Objekt weiter. Für den *XYDuplicateEliminator* unterscheiden sich zwei *PointSet*-Objekte genau dann, wenn ein Objekt einen Punkt mit einer X-Y-Koordinate enthält, für den kein Punkt mit der gleichen X-Y-Koordinate im anderen Objekt existiert. Damit ist sichergestellt, dass der nachfolgende `Triangulator`-Operator nur dann ein neues Dreiecksnetz berechnet, wenn sich die Position eines Sensors geändert hat, oder wenn ein Sensor hinzugekommen oder weggefallen ist. Das Dreiecksnetz wird mit Hilfe eines *TriangleSet*-Objektes dargestellt. Der *Triangulator* operiert für die Berechnung des Dreiecksnetzes ausschließlich auf den X- und Y-Koordinaten der Punkte. Die dritte Koordinate, welche die skalierte Temperatur enthält, bleibt unangetastet. Damit ist das Höhenfeld gebildet.

Das Höhenfeld wird nun an den `VertexSetUpdater` und den *AttributeAdder* gesendet. Der `AttributeAdder` fügt dem *TriangleSet*-Objekt noch nicht enthaltene Attribute hinzu. Dies können zum Beispiel eine Farbe, die Reflexionseigenschaften der Farbe oder viele andere Eigenschaften sein. Das so gebildete Dreiecksnetz wird zu der Rendering-Komponente geschickt und dort angezeigt.

Bis jetzt ist das angezeigte Höhenfeld statisch, die aktuellen Sensorwerte sind noch nicht in diesem Objekt reflektiert. Hierfür zu sorgen, ist Aufgabe des `VertexSetUpdater`-Operators. Er hält das aktuell von der Rendering-Komponente angezeigte *TriangleSet*-Objekt in seinem Speicher. Gleichzeitig bekommt er auch alle *PointSet*-Objekte des `ExpressionMappers`. Diese enthalten genau einen Punkt, dessen Z-Koordinate den geänderten, skalierten Temperaturwert enthält. Der `VertexSetUpdater` muss nun nur den aktuell empfangenen Punkt mit dem *TriangleSet*-Objekt in seinem Cache vergleichen. Enthält das *TriangleSet*-Objekt einen Punkt mit der gleichen X-Y-Koordinate, so wird ein *UpdateCommand* generiert. Dieses enthält den

Punkt, mit dem das *TriangleSet*-Objekt in der Rendering-Komponente aktualisiert werden soll, und den Index des Punktes innerhalb des *TriangleSet*-Objekts und die NOL des zu aktualisierenden *TriangleSet*-Objekts.

Die Rendering-Komponente muss nun das derzeit angezeigte *TriangleSet*-Objekt mit Hilfe der empfangenen *UpdateCommand*-Objekte aktualisieren. Damit wird das Höhenfeld dynamisch. Die Höhen der Punkte des unterliegenden Dreiecksnetzes ändern sich abhängig von den gesendeten Temperaturwerten. Die Triangulation findet nur dann statt, wenn diese auch benötigt wird, also wenn sich dass unterliegende Dreiecksnetz beim Wegfallen, Hinzukommen oder Positionänderungen der Sensorkomponenten ändert.

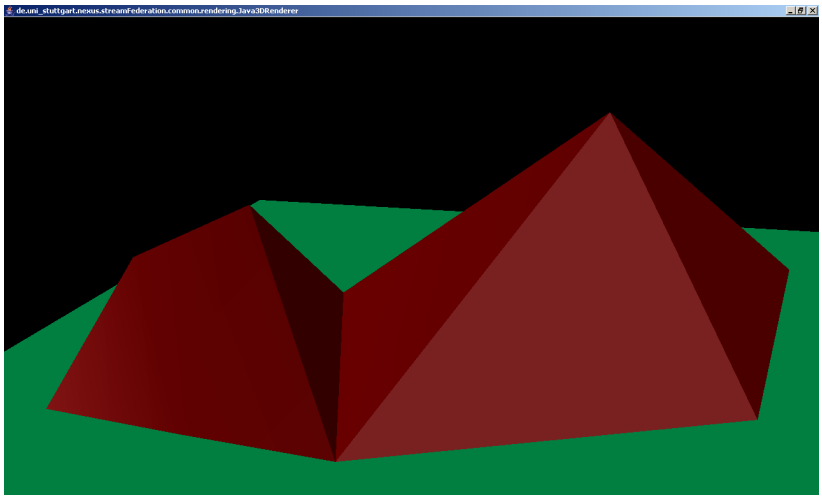


Abbildung 6.7.: Visualisierung von skalaren Werten mit Hilfe eines Höhenfeldes

Abbildung 6.7 zeigt die Visualisierung von skalaren Datenwerten mit Hilfe eines Höhenfelds. Zusehen sind zwei Anhebungen, die jeweils einen hohen Datenwert repräsentieren.

Kapitel 7

Zusammenfassung

7.1. Schlussfolgerung

In dieser Arbeit wurden Ansätze zur Realisierung einer verteilten, virtualisierten Visualisierungspipeline mit Hilfe der NEXUS Datenstrom-Föderation vorgestellt. Hierfür wurden zuerst die benötigten Grundlagen im Bereich der Visualisierung und der strombasierten Datenverarbeitung gelegt. Anschließend wurde das Umfeld und die Architektur der Ausführungsumgebung, der NEXUS Datenstrom-Föderation, beschrieben. Nach der grundsätzlichen Untersuchung, welche Komponenten und welche Funktionalität in einem solchen Visualisierungssystem benötigt werden, wurden Prototypen verschiedener Komponenten entwickelt, mit deren Hilfe eine verteilte, virtualisierte Visualisierungspipeline realisiert werden kann. Die Messungen der Bildraten und der Latenzen der Komponenten ergaben, dass zumindest mit leistungsfähigen Endgeräten in lokalen Netzwerken, in denen geringe Latenzen entstehen und hohe Bandbreiten zur Verfügung stehen, eine interaktive Visualisierung stattfinden kann. Die Vorstellung zweier Visualisierungsszenarien beschreiben beispielhaft die Ausführungsanordnungen verschiedener Operatoren, welche das konkrete Szenario realisieren.

Die entwickelten Komponenten können verwendet werden, um auf generische Art und Weise beliebige, abstrakte Daten zu visualisieren. Jedoch zeigten die vorgestellten Visualisierungsszenarios, dass für die intelligente, ressourcensparende Visualisierung häufig auf einen konkreten Anwendungsfall spezialisierte Operatoren benötigt werden. Es bleibt abzuwarten, welche Teile in Zukunft direkt von der NEXUS-Plattform direkt unterstützt werden, und welche Teile vom Benutzer durch die Definition eigener Operatoren abgedeckt werden.

Viele der Komponenten lassen eine dynamische, kontextabhängige Steuerung durch das Setzen von Parametern zu. Die Rendering-Komponenten unterstützen darüber hinaus verschiedene Techniken, die die Navigation in dem visualisierten Szenario zu ermöglichen. Jedoch wird für den zukünftigen Einsatz in einem Produktivsystem eine Anfragesprache und ein zugehöriger Anfrageprozessor benötigt, um die einfache Definition von Visualisierungsanfragen zu ermöglichen.

7.2. Entwicklungsmöglichkeiten

Die weiteren Entwicklungsmöglichkeit des hier entworfenen Prototyps sind vielseitig und hängen von den vielschichtigen Anforderungen ab. Konkreter Bedarf zur Integration besteht natürlich bei „echten“ Sensor-Komponenten, die während dieser Arbeit nur simuliert wurden. Weiterhin sollten bestehende NEXUS-Komponenten in die Datenstrom-Föderation integriert werden, um die schon vorhandene Funktionalität der NEXUS-Plattform auch für Datenströme nutzen zu können. Dies betrifft vor allem die Spatial Model Server, welche die statischen Elemente des Kontextmodells speichern. Natürlich gibt es viele weitere Entwicklungsfelder. Im Folgenden werden einige detailliert beschrieben.

7.2.1. Nexus-Datenmodell

Wie im Zuge dieser Arbeit gezeigt, konnte das NEXUS-Datenmodell verwendet werden, um einige visuelle Objekte, wie sie in anderen Modellierungssprachen auch vorkommen können, zu modellieren. Allerdings musste auch festgestellt werden, dass die Beschränkung auf eine Hierarchie von zwei Attribut-Ebenen das Kombinieren von verschiedenen Geometrien zu einem einzigen visuellen Objekt unmöglich macht. Eine Erweiterung des Datenmodells, so dass beliebig komplexe Objekte modelliert werden können, wäre hier von Nöten.

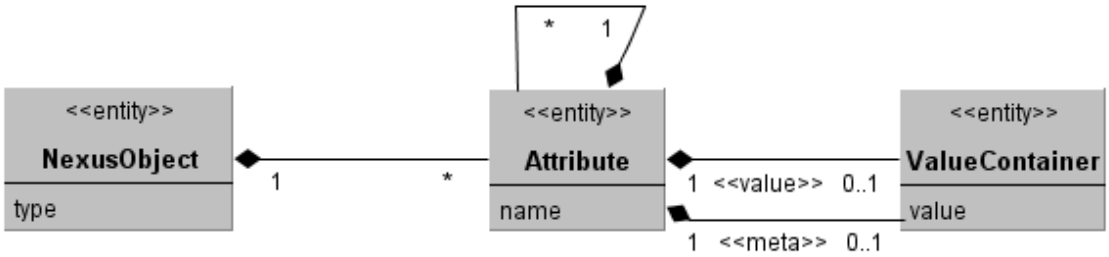


Abbildung 7.1.: Vorgeschlagenes Datenmodell

Abbildung 7.1 zeigt das hier vorgeschlagene Datenmodell. Anstatt Attribute als Menge von Attributteilen aufzufassen, geht dieses Modell davon aus, dass Attribute Mengen von weiteren Attributen sind. Ein Attribut kann nun einen Attributwert (*ValueContainer*) besitzen oder aus mehreren anderen Attributen zusammengesetzt sein, oder möglicherweise beides. Ein derartiges Modell ließe sich wie das bisherige in XML übersetzen (siehe Beispiel 7.1). Eine Referenzierung der Attribute und Attributwerte wären mit Pfadangaben nach dem Prinzip von XPath [W3C07] möglich. Dies entspricht dem aktuell verwendeten Prinzip der Referenzierung innerhalb von AWQL-Anfragen. Hier wären also keine Anpassungen nötig.

```
<nexusobject>
        <attribute_1>
                <value>blub</value><meta>bla</meta>
                <attribute_1_1>
```

```
                <value>blub</value><meta>bla</meta>
                <attribute_1_1_1>...</attribute_1_1_1>
                ...
            </attribute_1_1>
            <attribute_1_2>...</attribute_1_2>
            ...
        </attribute_1>
        <attribute_2>...</attribute_2>
        ...
</nexusobject>
```

Beispiel 7.1: Übersetzung des vorgeschlagenen Datenmodells ins XML-Format

7.2.2. Erweiterung um die DSMS-Funktionalität

Diese Arbeit geht davon aus, dass für die Datenselektion der Visualisierungspipeline in ferner Zukunft die Funktionalität eines DSMS bereitsteht. Hierfür müssen notwendige Operatoren entwickelt werden, Konzepte zur Integration der Zeit-Komponente in Anfragen erarbeitet werden und Möglichkeiten zur näherungsweisen Anfragebearbeitung gefunden werden. Inwiefern der Föderations-Manager und die Anfragesprache, die hierfür entwickelt werden, auch dafür verwendet werden können, um die Funktionalität der hier entwickelten Visualisierungspipeline abzubilden, bleibt abzuwarten. Ob es eine einzige generische Anfragesprache für alle Anwendungsgebiete geben wird, oder domänenspezifischen Anfragesprachen und Anfrageprozessoren zugelassen werden, ist noch zu diskutieren. Die Ergebnisse dieser Arbeit lassen den Schluss zu, dass ein Föderations-Manager, der die Verwaltung einer virtualisierten und verteilten Visualisierungspipeline übernimmt, sehr spezielle Aufgaben erfüllen muss, die sich vor allem was Optimierungsstrategien anbelangt deutlich von einem DSMS unterscheiden. Deswegen wird hier empfohlen, die Komponenten der einzelnen Anwendungsfelder unabhängig voneinander zu entwickeln und damit verschieden Anfragesprachen und Anfrageprozessoren zuzulassen.

7.2.3. Binärrepräsentation von GenericObjects

Die Ergebnisse der Evaluation zeigen, dass bei der Serialisierung und Deserialisierung Flaschenhälse entstehen können, wenn komplexere visuelle Objekte gesendet werden. Beispiel 7.2 zeigt den Umfang der Definition eines einzelnen Punktes. Bedenkt man, dass visuelle Objekte mit Leichtigkeit tausende solcher Punkt enthalten können, so springt es geradezu ins Auge, dass hier noch Potential zur Optimierung besteht. Die Dateigröße von Beispiel 7.2 beläuft sich auf 262 Bytes. Die enthaltene Informationen sind drei Ganzzahl-Werte, welche die Koordinaten enthalten, und ein Ganzzahl-Wert, der das verwendete Koordinatensystem spezifiziert. Mit einem binären Format und 4 Byte großen Ganzzahl-Typen wäre Information prinzipiell in auch mit $4 \times 4 = 16$ Byte darstellbar.

```
<eas:point>
        <nsas:value>
            <eas:pos>
```

```
                    <gml:Point srsName="http://www.opengis.net/gml/srs/epsg.xml#40000">
                          <gml:coordinates>0.5,0.0,0.0</gml:coordinates>
                    </gml:Point>
              </eas:pos>
        </nsas:value>
</eas:point>
```

Beispiel 7.2: Definition eines Punktes mit der hier entwickelten Modellierungsprache

Damit entsteht der Bedarf an einem binären Übertragungsformat für *NexusObject*s. Es wird jedoch nicht empfohlen, ein neues hocheffizientes Übertragungsformat zu entwickeln, welches nur für die visuellen Objekte spezifiziert wurde. Der Gedanke eines generischen, anwendungsunabhängigen Datenmodells sollte beibehalten werden. Es ist zu überprüfen, ob das NEXUS-Datenmodell in ein effizienteres Übertragungsformat übersetzt werden kann. Hierbei ist nicht nur darauf zu achten, die Informationen möglichst kompakt zu übermitteln, sondern vor allem darauf, wie schnell aus der serialisierten Form die benötigten Information gewonnen, bzw. wie schnell die Informationen in das serialisierte Format gebracht werden können.

7.2.4. Entwicklung eines blockierenden Sende-Mechanismus

Bei der Entwicklung der Bildgenerierungs-Komponente ist aufgefallen, dass der von der Ausführungsumgebung angebotene Sende-Mechanismus nicht ausreichend ist. Der Prozess der Bildgenerierung läuft bei höheren Auflösungen mit einer höheren Rate als der Prozess der JPEG-Komprimierung. Dies hat zur Folge, dass sich die Warteschlange zwischen den Operatoren zuerst auffüllt und dann überläuft. Zwar hat der Datenverlust bei hohen Raten keinen sichtbaren Effekt, jedoch sollte er aus Effizienzgründen vermieden werden. Außerdem besteht das Problem unabhängig von der Bildgenerierung und könnte auch in einem anderen Kontext auftreten, in dem Datenverlust nicht tolerierbar ist. Da es von der Semantik der zu realisierenden Anfrage abhängt, ob Datenverlust tolerierbar ist oder nicht, sollte die Möglichkeit bestehen, das Sendeverhalten der Komponenten zur Laufzeit festzulegen.

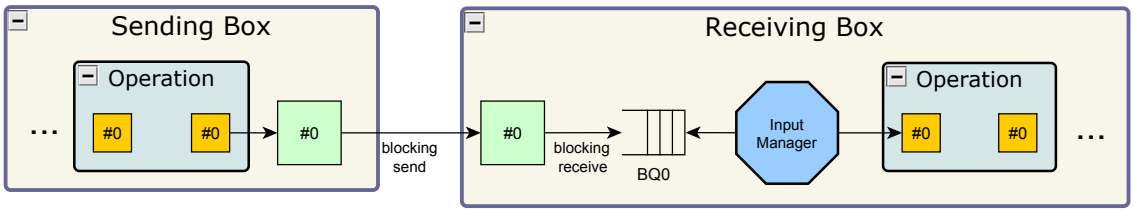


Abbildung 7.2.: Blockierendes Sendeverhalten in einer Box-Realisierung

Das Sendeverhalten eines Operators wird durch das Empfangsverhalten der Realisierung der empfangenen `Box`-Schnittstelle festgelegt. Abbildung 7.2 zeigt, wie eine Implementierung mit einer blockierenden Empfangsmethode (`receive(int, Object)`) würde letzten Endes die Sende-Methode (`send(Object,int)`), des sendenden Operators blockieren. Das gut

gelungene Design der Datenstrom-Föderation nach [Sar08] lässt eine einfache Integration einer `Box`-Realisierung mit anderer Semantik zu. Diese Implementierung ist momentan noch nicht innerhalb einer Setup-Datei spezifizierbar und damit besteht keine Möglichkeit, diese zur Laufzeit festzulegen.

7.2.5. Weiterentwicklung der Datenstrom-Föderation zu einer Medien-Plattform

Als Videoübertragungsformat wurde für diese Arbeit ein Strom aus JPEG-Bildern verwendet. Der Grund hierfür waren die noch akzeptablen Latenzen bei der Kodierung und Dekodierung, sowie die Verringerung der zu übertragenden Daten gegenüber der unkomprimierten Übertragung. Für viele Fälle stellt dies sicherlich einen annehmbaren Kompromiss dar. Es sind aber auch Anwendungsfälle denkbar, bei denen es nicht auf eine geringe Latenz ankommt, sondern auf möglichst geringe Datenraten. Dies ist der Fall, wenn kein Bedarf zur Interaktion besteht und ein Netzwerk verwendet wird, in dem geringe Bandbreiten vorherrschen. Das Standardbeispiel hierfür ist das Internet. Hier stehen in der Regel für den Einzelbenutzer geringere Übertragungsraten zur Verfügung. Im Übrigen können hier die kommunizierenden Rechner geographisch sehr weit voneinander entfernt liegen. Deshalb können wegen der Signallaufzeit und der Vermittlungszeit im paketvermittelten IP-Netzwerk schon so hohe Latenzen entstehen, dass keine unmittelbare Interaktion mehr möglich wäre. Deshalb tangiert es in diesen Fällen häufig nur peripher, wenn die Latenz durch ein effizienteres Kodierungsverfahren noch weiter ansteigt. Demnach sollte die Datenstrom-Föderation auch andere Videoübertragungsformate, wie zum Beispiel MPEG-4 [ISO04a] unterstützen. Dies ist natürlich nicht nur auf Videoformate beschränkt. Nach der Definition der Visualisierung von [FR94] sind auch andere Wahrnehmungsarten wie die auditive Wahrnehmung im Begriff der Visualisierung eingeschlossen. Somit sollte es auch möglich sein, Audioströme zu übertragen. Einher geht die Frage, wie zwischen den verschiedenen Strömen, zum Beispiel Audio und Video, synchronisiert werden kann. Hier kommt Forderung auf, verschiedene, so genannte *Real-Time-Protokolle*, zu unterstützen, wie zum Beispiel das *Real-Time Transport Protocol (RTP)* [IET03], oder in der IP-Telefonie verwendete Protokoll *H.323* [IT06]. Es bleibt die Frage, ob für eine Java-Laufzeit-Umgebung in Zukunft eine Medien-Plattform zur Verfügung steht, die die Funktionalität dieser Anwendungsdomäne bietet. Das von Sun entwickelte JMF, von welchem für diese Arbeit die Abspiel-Funktionaliät verwendet wurde, ist auf Grund veralteter Codecs unbrauchbar. Vielversprechend ist das von Sun gepushte JavaFX-Projekt, welches gerade im Enstehen ist. JavaFX ist skriptbasiert, es gibt jedoch auch die Möglichkeit in einer herkömmlichen Java-Klasse die Funktionalität von JMF zu nutzen. Wie sich dieses Projekt weiterentwickelt und ob es verwendet werden kann, um die Datenstrom-Föderation medienfähig zu machen, muss noch untersucht werden. Jedenfalls zeigen aktuelle Demos die Unterstützung aktueller Codecs wie MPEG-4.

Für die Datenstrom-Föderation hat das Einbeziehen der Medien-Konzepte wahrscheinlich mehrere Implikationen. Für die Unterstützung von Real-Time-Protokollen müssen eventuell die vorhanden Transport-Mechanismen angepasst werden. Bei der Real-Time-Übertragung

ist häufig eine unzuverlässige Verbindung mit geringer Latenz einer zuverlässigen Verbindung mit hoher Latenz vorzuziehen. Denn kommen Pakete erst nach dem Abspielzeitpunkt beim Empfänger an, so sind diese nutzlos und müssen verworfen werden.

Weiterhin ist es sinnvoll, in diesem Zuge ein neues Verarbeitungsmodell der Operatoren einzuführen. Aktuell basiert das Verarbeitungsmodell auf passiven Operatoren, die Daten nach dem Nachrichten-Paradigma verarbeiten. Die Nachrichten sind abgeschlossenen Einheiten, nämlich beliebige `Object`-Instanzen, die einem Operator von einem `InputManager` durch Aufruf der `process(inputID int, Object data)`-Methode übergeben werden. Dies ist nicht die passende Programmierabstraktion für die Verarbeitung von Medien. In der Regel sind die inneren Strukturen eines MPEG-4-Stroms dem Anwendungsentwickler nicht bekannt. Er benutzt für die Übermittlung und Kodierung eine API, die diese Details nach außen verbirgt. Damit wäre eine passende Programmierabstraktion für beliebige Audio- oder Videoströme die eines transparenten Bitstroms. Um Bitströme zu verarbeiten, wäre ein Verarbeitungsmodell von aktiven Operatoren nötig, deren Ein- und Ausgänge aus `InputStream`-Instanzen bzw. `OutputStream`-Instanzen bestehen.[1] Während mit dem aktuellen Verarbeitungsmodell der *InputManager* den Zeitpunkt der Verarbeitung einer Dateneinheit bestimmt, so könnten die aktiven Operatoren den Zeitpunkt selbst bestimmen, wann von einem `InputStream` Daten gelesen werden. Das Verarbeitungsmodell eines aktiven Operators ließe sich ohne große Veränderung in das bestehende Verarbeitungsmodell einarbeiten. Die grundlegende Idee hierbei ist, nach außen hin das Nachrichten-basierte Verarbeitungsmodell weiterhin zu unterstützen, während nach innen hin das Verarbeitungsmodell eines transparenten Bitstroms angeboten wird. Abbildung 7.3 zeigt das prinzipielle Vorgehen.

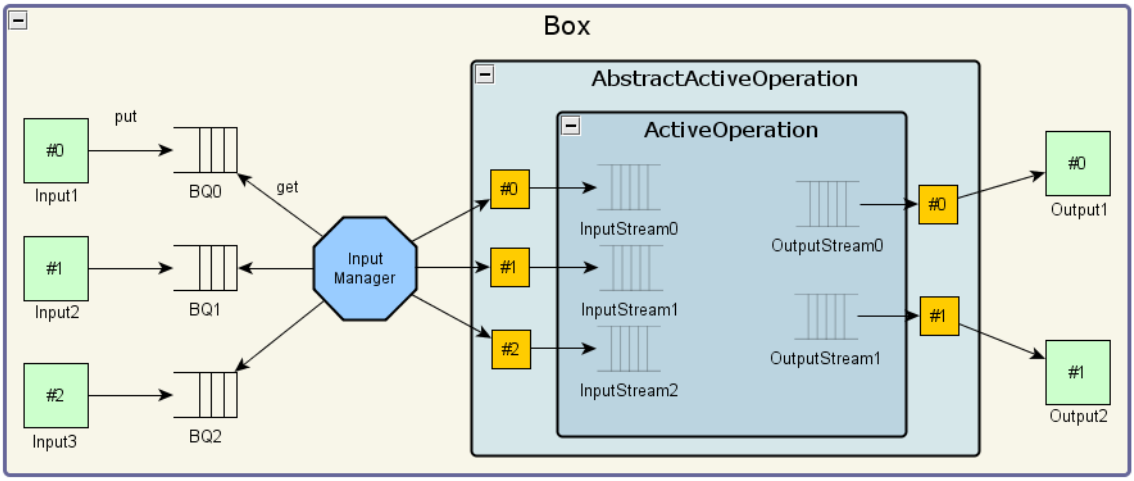


Abbildung 7.3.: Aktive Operation-Realisierung

Das heißt, es wäre notwendig, eine abstrakte Superklasse aller aktiven Operatoren zu entwickeln, die hier im Folgenden `AbstractActiveOperation` genannt wird. Diese würde von

[1] Ein `InputStream` bzw. `OutputStream` ist eine Java-Schnittstelle, die die Verarbeitung von Bitströmen ermöglicht. Zur Verarbeitung stehen Lese- bzw. Schreibe-Methoden zur Verfügung.

außen Byte-Arrays empfangen, und die empfangenen Byte-Arrays der `InputSream`-Instanz übergeben, welche zur Implementierung des aktiven Operators hin Lese-Methoden zur Verfügung stellt. Der aktive Operator könnte nun bei Bedarf von der `InputStream`-Instanz lesen und eventuelle Ergebnisse auf eine `OutputStream`-Instanz schreiben. Die `OutputStream`-Instanz wiederum stellt die geschriebenen Bytes der `AbstractActiveOperation` zur Verfügung, welche dafür zuständig diese in gewissen Byte-Array-Blöcken weiter zu senden. Die Unterteilung in Byte-Array-Blöcke würde dann allerdings unabhängig von den Daten, beispielsweise abhängig von einer Maximum Transmission Unit (MTU), stattfinden. Dies ist nicht immer sinnvoll, da die enthaltenen Daten nicht in semantisch sinnvollen Einheiten übermittelt werden. Wird eine semantische Einheit auf mehrere Datenblöcke verteilt und geht einer dieser Datenblöcke bei der Übermittlung verloren, sind alle übrigen Datenblöcke nutzlos und müssen verworfen werden. Wird jedoch eine zuverlässige Verbindung benutzt, die Datenverlust ausschließt, kann dieses Verfahren verwendet werden.

Anhang A

VisualClassSchema

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:ecs="http://www.nexus.uni-stuttgart.de/1.0/Visual/ECS"
    xmlns:eas="http://www.nexus.uni-stuttgart.de/1.0/Visual/EAS"
    xmlns:nscs="http://www.nexus.uni-stuttgart.de/1.0/NSCS"
    xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
    targetNamespace="http://www.nexus.uni-stuttgart.de/1.0/Visual/ECS"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
       <annotation>
              <documentation>An extended class schema forVisual Objects.</documentation>
       </annotation>
       <import namespace="http://www.nexus.uni-stuttgart.de/1.0/Visual/EAS"
           schemaLocation="VisualAttributeSchema.xsd"/>

       <complexType name="Command">
              <annotation>
                     <documentation>An operator command. </documentation>
                     <appinfo>
                            <extension base="nscs:NexusDataObject"/>
                     </appinfo>
              </annotation>
              <complexContent>
                     <extension base="nscs:NexusDataObject"/>
              </complexContent>
       </complexType>

       <complexType name="CacheCommand">
              <annotation>
                     <documentation>An operator command. </documentation>
                     <appinfo>
                            <extension base="ecs:Command"/>
                     </appinfo>
              </annotation>
              <complexContent>
                     <extension base="ecs:Command"/>
              </complexContent>
       </complexType>

       <complexType name="FlushCommand">
```

```
        <annotation>
                <documentation>An operator command. </documentation>
                <appinfo>
                        <extension base="ecs:Command"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Command"/>
        </complexContent>
</complexType>

<complexType name="DeleteCommand">
        <annotation>
                <documentation>A delete command.
          "restrictionNo1" selects the object you want to delete.
          </documentation>
                <appinfo>
                        <extension base="ecs:Command"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Command">
                        <sequence>
                                <element ref="eas:restrictionNo1"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="UpdateCommand">
        <annotation>
                <documentation>An update command.
          "restrictionNo1" selects the object you want to update.
          "targetAttribute" selects the attribute part to update
          "index" specifies the index of the attribute to update (since attributes
              can be lists)
          "updateValue" specifies the value of the attribute part to
              update</documentation>
                <appinfo>
                        <extension base="ecs:Command"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Command">
                        <sequence>
                                <element ref="eas:restrictionNo1"/>
                                <element ref="eas:update" maxOccurs="unbounded"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="RenderingCommand">
        <annotation>
```

```
                <documentation>The RenderingCommand can be used to change some
                    parameters in the rendering component</documentation>
                <appinfo>
                        <extension base="ecs:Command"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Command">
                        <sequence>
                                <element ref="eas:viewTranslation"/>
                                <element ref="eas:viewRotation"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="MappingRule">
        <annotation>
                <documentation>Mapping rules for the ExpressionMapper</documentation>
                <appinfo>
                        <extension base="nscs:NexusDataObject"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="nscs:NexusDataObject">
                        <sequence>
                                <element ref="eas:targetAttribute" minOccurs="0"
                                    maxOccurs="unbounded"/>
                                <element ref="eas:destAttribute" minOccurs="0"
                                    maxOccurs="unbounded"/>
                                <element ref="eas:expression" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="VisualObject">
        <annotation>
                <documentation>a visual object</documentation>
                <appinfo>
                        <extension base="nscs:NexusDataObject"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="nscs:NexusDataObject">
                        <sequence>
                                <element ref="nsas:pos"/>
                                <element ref="eas:posInterpolation" minOccurs="0"/>
                                <element ref="eas:rotation" minOccurs="0"/>
                                <element ref="eas:rotationInterpolation" minOccurs="0"/>
                                <element ref="eas:refNol" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
```

```
</complexType>

<complexType name="Light">
        <annotation>
                <documentation>light</documentation>
                <appinfo>
                        <extension base="nscs:NexusDataObject"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="nscs:NexusDataObject">
                        <sequence>
                                <element ref="nsas:color" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="AmbientLight">
        <annotation>
                <documentation>Ambient light</documentation>
                <appinfo>
                        <extension base="ecs:Light"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Light"/>
        </complexContent>
</complexType>

<complexType name="PointLight">
        <annotation>
                <documentation> A light source which radiates form a specified
                    point</documentation>
                <appinfo>
                        <extension base="ecs:Light"/>
                </appinfo>
                <documentation>a visual object</documentation>
        </annotation>
        <complexContent>
                <extension base="ecs:Light">
                        <sequence>
                                <element ref="nsas:pos" minOccurs="0"/>
                                <element ref="eas:attenuation" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="SpotLight">
        <annotation>
                <documentation>Light which radiates from a specified point with a
                    specified angle</documentation>
                <appinfo>
                        <extension base="ecs:PointLight"/>
```

```
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:PointLight">
                        <sequence>
                                <element ref="eas:direction" minOccurs="0"/>
                                <element ref="eas:spreadAngle" minOccurs="0"/>
                                <element ref="eas:concentration" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="DirectionalLight">
        <annotation>
                <documentation>Models parallel light rays</documentation>
                <appinfo>
                        <extension base="ecs:Light"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Light">
                        <sequence>
                                <element ref="eas:direction" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="Shape">
        <annotation>
                <documentation>Super class of all shapes. </documentation>
                <appinfo>
                        <extension base="ecs:VisualObject"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:VisualObject">
                        <sequence>
                                <element ref="eas:appearance" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="Box">
        <annotation>
                <documentation>A Box centered at origin with a length in x, y and z
                    direction</documentation>
                <appinfo>
                        <extension base="ecs:Shape"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Shape">
```

```
                        <sequence>
                                <element ref="eas:x" minOccurs="0"/>
                                <element ref="eas:y" minOccurs="0"/>
                                <element ref="eas:z" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="Sphere">
        <annotation>
                <documentation>A Sphere centered at origin with a
                    radius</documentation>
                <appinfo>
                        <extension base="ecs:Shape"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Shape">
                        <sequence>
                                <element ref="eas:radius" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="Cone">
        <annotation>
                <documentation>A Cone centered at origin with a radius and a
                    height</documentation>
                <appinfo>
                        <extension base="ecs:Shape"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Shape">
                        <sequence>
                                <element ref="eas:radius" minOccurs="0"/>
                                <element ref="eas:height" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="Cylinder">
        <annotation>
                <documentation>A Cylinder centered at origin with a radius and a
                    height</documentation>
                <appinfo>
                        <extension base="ecs:Shape"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:Shape">
                        <sequence>
```

```
                        <element ref="eas:radius" minOccurs="0"/>
                        <element ref="eas:height" minOccurs="0"/>
                    </sequence>
                </extension>
            </complexContent>
        </complexType>

        <complexType name="VertexSet">
            <annotation>
                <documentation>Super class of all classes containing a set of
                    points</documentation>
                <appinfo>
                    <extension base="ecs:Shape"/>
                </appinfo>
            </annotation>
            <complexContent>
                <extension base="ecs:Shape">
                    <sequence>
                        <element ref="eas:point" minOccurs="0"
                            maxOccurs="unbounded"/>
                    </sequence>
                </extension>
            </complexContent>
        </complexType>

        <complexType name="PointSet">
            <annotation>
                <documentation>A set of colored points</documentation>
                <appinfo>
                    <extension base="ecs:VertexSet"/>
                </appinfo>
            </annotation>
            <complexContent>
                <extension base="ecs:VertexSet"/>
            </complexContent>
        </complexType>

        <complexType name="LineSet">
            <annotation>
                <documentation>A set of colored Lines</documentation>
                <appinfo>
                    <extension base="ecs:VertexSet"/>
                </appinfo>
            </annotation>
            <complexContent>
                <extension base="ecs:VertexSet"/>
            </complexContent>
        </complexType>

        <complexType name="TriangleSet">
            <annotation>
                <documentation>A set of colored Triangles</documentation>
                <appinfo>
                    <extension base="ecs:VertexSet"/>
                </appinfo>
```

```
        </annotation>
        <complexContent>
                <extension base="ecs:VertexSet"/>
        </complexContent>
</complexType>

<complexType name="QuadSet">
        <annotation>
                <documentation>A set of colored Quads</documentation>
                <appinfo>
                        <extension base="ecs:VertexSet"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:VertexSet"/>
        </complexContent>
</complexType>

<complexType name="PolygonSet">
        <annotation>
                <documentation>A set of colored Polygons</documentation>
                <appinfo>
                        <extension base="ecs:VertexSet"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:VertexSet">
                        <sequence>
                                <element ref="eas:stripCount" minOccurs="0"
                                    maxOccurs="unbounded"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="TexturedPlane">
        <annotation>
                <documentation>A plane which is textured with an image. Could be used
                    to visualize maps</documentation>
                <appinfo>
                        <extension base="ecs:VisualObject"/>
                </appinfo>
        </annotation>
        <complexContent>
                <extension base="ecs:VisualObject">
                        <sequence>
                                <element ref="eas:texture" minOccurs="0"/>
                                <element ref="eas:vertices" minOccurs="0"/>
                        </sequence>
                </extension>
        </complexContent>
</complexType>

<complexType name="GenericVisualObject">
        <annotation>
```

```
                <documentation>Provides generic means to define a visual object via a
                    modelling language</documentation>
                <appinfo>
                        <extension base="ecs:VisualObject"/>
                </appinfo>
            </annotation>
            <complexContent>
                <extension base="ecs:VisualObject">
                        <sequence>
                                <element ref="eas:model" minOccurs="0"/>
                        </sequence>
                </extension>
            </complexContent>
        </complexType>
</schema>
```

Anhang B

VisualAttributeSchema

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:eas="http://www.nexus.uni-stuttgart.de/1.0/Visual/EAS"
    xmlns:nsas="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
    xmlns:nsat="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
    attributeFormDefault="unqualified" elementFormDefault="qualified"
    targetNamespace="http://www.nexus.uni-stuttgart.de/1.0/Visual/EAS">
       <import namespace="http://www.nexus.uni-stuttgart.de/1.0/NSAS"
           schemaLocation="NexusStandardAttributeSchema.xsd"/>
       <import namespace="http://www.nexus.uni-stuttgart.de/1.0/NSAT"
           schemaLocation="NexusStandardAttributeTypes.xsd"/>

       <complexType name="NexusPosInterpolationType">
              <annotation>
                     <documentation>A complex Nexus Attribute Type that allows the
                         interpolation between positions</documentation>
              </annotation>
              <sequence>
                     <element name="value">
                            <complexType>
                                   <sequence>
                                          <element minOccurs="0" name="newPos"
                                              type="nsat:NexusPointType"/>
                                          <element minOccurs="0" name="duration"
                                              type="nsat:NexusIntegerType"/>
                                          <element minOccurs="0" name="numberOfLoops"
                                              type="nsat:NexusIntegerType"/>
                                   </sequence>
                            </complexType>
                     </element>
                     <element minOccurs="0" ref="nsas:meta"/>
              </sequence>
       </complexType>

       <complexType name="NexusEncodedDataAttributeType">
              <annotation>
                     <documentation>The content type attribute declares the type of the
                         binary data. It uses the content-type specifications of MIME
                         </documentation>
```

```
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="1"
                                            name="encoding" type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="contentType"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="1" name="data"
                                            type="nsat:NexusStringType"/>
                                </sequence>
                        </complexType>
                </element>
                <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<complexType name="VisualPointAttributeType">
        <annotation>
                <documentation/>
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="1" name="pos"
                                            type="nsat:NexusPointType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="color" type="nsat:NexusStringType"/>
                                </sequence>
                        </complexType>
                </element>
                <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<complexType name="NexusRotationAttributeType">
        <annotation>
                <documentation/>
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="0" name="x"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="y"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="z"
                                            type="nsat:NexusFloatType"/>
                                </sequence>
                        </complexType>
                </element>
```

```
            <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<complexType name="NexusRotationInterpolationAttributeType">
        <annotation>
            <documentation/>
        </annotation>
        <sequence>
            <element name="value">
                <complexType>
                    <sequence>
                        <element maxOccurs="1" minOccurs="0" name="x"
                            type="nsat:NexusFloatType"/>
                        <element maxOccurs="1" minOccurs="0" name="y"
                            type="nsat:NexusFloatType"/>
                        <element maxOccurs="1" minOccurs="0" name="z"
                            type="nsat:NexusFloatType"/>
                        <element minOccurs="0" name="duration"
                            type="nsat:NexusIntegerType"/>
                        <element minOccurs="0" name="numberOfLoops"
                            type="nsat:NexusIntegerType"/>
                    </sequence>
                </complexType>
            </element>
            <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>


<complexType name="AppearanceAttributeType">
        <annotation>
            <documentation>Defines appearance attributes.</documentation>
        </annotation>
        <sequence>
            <element name="value">
                <complexType>
                    <sequence>
                        <element maxOccurs="1" minOccurs="0"
                            name="coloring_color"
                            type="nsat:NexusStringType"/>
                        <element maxOccurs="1" minOccurs="0"
                            name="coloring_shadeModel"
                            type="nsat:NexusStringType"/>
                        <element maxOccurs="1" minOccurs="0"
                            name="line_pattern"
                            type="nsat:NexusStringType"/>
                        <element maxOccurs="1" minOccurs="0"
                            name="line_patternMask"
                            type="nsat:NexusStringType"/>
                        <element maxOccurs="1" minOccurs="0"
                            name="line_patternScaleFactor"
                            type="nsat:NexusFloatType"/>
```

```
                                        <element maxOccurs="1" minOccurs="0"
                                            name="line_antialiasing"
                                            type="nsat:NexusBooleanType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="line_width"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_ambientColor"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_diffuseColor"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_emissiveColor"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_specularColor"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_lighting"
                                            type="nsat:NexusBooleanType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_shininess"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="material_colorTarget"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="point_size"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="point_antialiasing"
                                            type="nsat:NexusBooleanType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="polygon_cullFace"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="polygon_backFaceNormalFlip"
                                            type="nsat:NexusBooleanType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="polygon_mode"
                                            type="nsat:NexusStringType"/>
                                    </sequence>
                                </complexType>
                            </element>
                            <element minOccurs="0" ref="nsas:meta"/>
                        </sequence>
                    </complexType>

                    <complexType name="NexusTargetAttributeAttributeType">
                        <annotation>
                            <documentation/>
                        </annotation>
                        <sequence>
                            <element name="value">
```

```
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="1" name="path"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="geoPointIndex"
                                            type="nsat:NexusIntegerType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="geoPointComponent"
                                            type="nsat:NexusIntegerType"/>
                                        <element maxOccurs="1" minOccurs="1"
                                            name="variable" type="nsat:NexusStringType"/>
                                </sequence>
                        </complexType>
                </element>
                <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<complexType name="NexusDestAttributeAttributeType">
        <annotation>
                <documentation/>
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="1" name="path"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="geoPointIndex"
                                            type="nsat:NexusIntegerType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="geoPointComponent"
                                            type="nsat:NexusIntegerType"/>
                                        <element maxOccurs="1" minOccurs="1"
                                            name="attrValue"
                                            type="nsat:NexusStringType"/>
                                </sequence>
                        </complexType>
                </element>
                <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<complexType name="NexusExpressionAttributeType">
        <annotation>
                <documentation/>
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="1"
                                            name="variable" type="nsat:NexusStringType"/>
```

```
                                        <element maxOccurs="1" minOccurs="1"
                                            name="varValue" type="nsat:NexusStringType"/>
                                    </sequence>
                            </complexType>
                    </element>
                    <element minOccurs="0" ref="nsas:meta"/>
            </sequence>
    </complexType>

    <complexType name="NexusUpdateAttributeType">
            <annotation>
                    <documentation/>
            </annotation>
            <sequence>
                    <element name="value">
                            <complexType>
                                    <sequence>
                                        <element maxOccurs="1" minOccurs="1"
                                            name="target" type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="index" type="nsat:NexusIntegerType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="updateValue"
                                            type="nsat:NexusStringType"/>
                                        <element maxOccurs="1" minOccurs="0"
                                            name="updateGeo"
                                            type="nsat:NexusGeometryType"/>
                                    </sequence>
                            </complexType>
                    </element>
                    <element minOccurs="0" ref="nsas:meta"/>
            </sequence>
    </complexType>

    <complexType name="NexusViewTranslationAttributeType">
            <annotation>
                    <documentation/>
            </annotation>
            <sequence>
                    <element name="value">
                            <complexType>
                                    <sequence>
                                        <element maxOccurs="1" minOccurs="0" name="x"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="y"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="z"
                                            type="nsat:NexusFloatType"/>
                                    </sequence>
                            </complexType>
                    </element>
                    <element minOccurs="0" ref="nsas:meta"/>
            </sequence>
    </complexType>
```

```
<complexType name="NexusViewRotationAttributeType">
        <annotation>
                <documentation/>
        </annotation>
        <sequence>
                <element name="value">
                        <complexType>
                                <sequence>
                                        <element maxOccurs="1" minOccurs="0" name="x"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="y"
                                            type="nsat:NexusFloatType"/>
                                        <element maxOccurs="1" minOccurs="0" name="z"
                                            type="nsat:NexusFloatType"/>
                                </sequence>
                        </complexType>
                </element>
                <element minOccurs="0" ref="nsas:meta"/>
        </sequence>
</complexType>

<element name="restrictionNol" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusNolAttributeType"/>
<element name="update" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusUpdateAttributeType"/>
<element name="viewTranslation" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusViewTranslationAttributeType"/>
<element name="viewRotation" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusViewRotationAttributeType"/>

<element name="model" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusEncodedDataAttributeType"/>
<element name="posInterpolation" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusPosInterpolationType"/>
<element name="rotation" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusRotationAttributeType"/>
<element name="rotationInterpolation" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusRotationInterpolationAttributeType"/>

<element name="texture" substitutionGroup="nsas:NexusAttribute"
    type="eas:NexusEncodedDataAttributeType"/>
<element name="vertices" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusGeometryAttributeType"/>
<element name="point" substitutionGroup="nsas:NexusAttribute"
    type="eas:VisualPointAttributeType"/>
<element name="stripCount" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusIntegerAttributeType"/>
<element name="appearance" substitutionGroup="nsas:NexusAttribute"
    type="eas:AppearanceAttributeType"/>
<element name="x" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusFloatAttributeType"/>
<element name="y" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusFloatAttributeType"/>
<element name="z" substitutionGroup="nsas:NexusAttribute"
    type="nsas:NexusFloatAttributeType"/>
```

```
        <element name="radius" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusFloatAttributeType"/>
        <element name="height" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusFloatAttributeType"/>
        <element name="color" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusStringAttributeType"/>
        <element name="direction" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusStringAttributeType"/>
        <element name="attenuation" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusStringAttributeType"/>
        <element name="concentration" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusFloatAttributeType"/>
        <element name="spreadAngle" substitutionGroup="nsas:NexusAttribute"
            type="nsas:NexusFloatAttributeType"/>
        <element name="targetAttribute" substitutionGroup="nsas:NexusAttribute"
            type="eas:NexusTargetAttributeAttributeType"/>
        <element name="destAttribute" substitutionGroup="nsas:NexusAttribute"
            type="eas:NexusDestAttributeAttributeType"/>
        <element name="expression" substitutionGroup="nsas:NexusAttribute"
            type="eas:NexusExpressionAttributeType"/>

</schema>
```

Abbildungsverzeichnis

2.1. Stufen der Visualisierungspipeline 15
2.2. Möglichkeiten der Verteilung einer Visualisierungspipeline 16
2.3. Datenmodell von Nexus 24
2.4. Nexus Standard Class Schema [NM01] 25
2.5. Nexus Architektur [Sar08] 27
2.6. Die NEXUS Datenstrom-Föderation (ähnl. [Sar08]). 28
2.7. Aufbau einer Box [Sar08] 29
2.8. Klassendiagramm einer Realisierung der `Operation`-Schnittstelle 30

3.1. Beispielhafte Umsetzung einer verteilten Visualisierungspipeline 37
3.2. Clientseitige Umsetzung der Bildgenerierung 39
3.3. Geometrieerzeugung als Funktion 40
3.4. Koordinatensysteme der Bilderzeugung 41
3.5. Transformationsverfahren [Sny87] 42
3.6. Klassenhierarchie der Command-Klasse 44
3.7. Versenden eines DeleteCommands 45
3.8. Klassenhierarchie der VisualObjects 50
3.9. Klassenhierarchie der Light-Klasse 54
3.10. Anfragesprachen unterschiedlicher Anwendungsfelder 57

4.1. Java3D Scene Graph [SRD00] 60
4.2. JMF Architektur [SM99] 63
4.3. Vereinfachtes Klassendiagramm der JMF-Komponenten 64

5.1. Gauß-Krüger Projektion 66
5.2. Klassendiagramm des CoordTransform-Operators 67
5.3. Gitter der zu interpolierenden Gitterpunkte 68
5.4. Funktionsweise des ExpressionMapper-Operators 70
5.5. Klassendiagramm Bildgenerierungs-Komponenten 71
5.6. Scenengraph des `Java3DRenderers` 72
5.7. Datenformat eines Rahmens 73
5.8. Klassendiagramm des AbstractPlatformSink 74
5.9. Videostrom-Verbindung zum Client 75

5.10. Video-Frame und Daten-Frame 75
5.11. Klassendiagramm der VideoViewer-Komponenten 76
5.12. Aktualisierungsrate des NAVGInterpolators 78
5.13. Rotierender Würfel 79
5.14. Bildrate in Abhängigkeit von der Auflösung 80
5.15. Instanzierungsdauer in Abhängigkeit zur Anzahl der Dreiecke 81
5.16. Dauer der Serialisisierung und Deserialisierung 82
5.17. Verarbeitungszeit verschiedener *Command*-Objekte 84
5.18. Wiedergaberate beim Abspielen 85
5.19. Aufteilung zur Messung der Latenz 88
5.20. Latenz 88

6.1. Operatorgraph zur Visualisierung von mobilen Objekten 92
6.2. Visualisierung von mobilen Objekten 94
6.3. Invalide Dreiecksnetze 95
6.4. Oberer Tangentenpunkt 96
6.5. Logischer Operatorgraph zur Visualisierung von Höhenfeldern 98
6.6. Arbeitsweise des `Union`-Operators 99
6.7. Visualisierung von skalaren Werten mit Hilfe eines Höhenfeldes 100

7.1. Vorgeschlagenes Datenmodell 102
7.2. Blockierendes Sendeverhalten in einer Box-Realisierung 104
7.3. Aktive Operation-Realisierung 106

Tabellenverzeichnis

2.1. Beispielanfragen an ein strombasiertes Visualisierungssystem 21

5.1. Verwendete Hardware-Systeme 77
5.2. Benötigte Übertragungsraten bei verschiedenen Auflösungen und Bildraten . 85

Literaturverzeichnis

[ABW03] ARASU, A. ; BABU, S. ; WIDOM, J.: The CQL continuous query language: Semantic foundations and query execution / Stanford University. 2003. – Forschungsbericht

[ACc+03] ABADI, Daniel J. ; CARNEY, Don ; ÇETINTEMEL, Ugur ; CHERNIACK, Mitch ; CONVEY, Christian ; LEE, Sangdon ; STONEBRAKER, Michael ; TATBUL, Nesime ; ZDONIK, Stan: Aurora: a new model and architecture for data stream management. In: *The VLDB Journal* 12 (2003), Nr. 2, S. 120–139

[ADK+99] AESCHLIMANN, M. ; DINDA, P. ; KALLIVOKAS, L. ; LOPEZ, J. ; LOWEKAMP, B. ; O'HALLARON, D.: Preliminary report on the design of a framework for distributed visualization. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniquesand Applications (PDPTA'99)*, 1999

[App05] APPLE, Inc.: *QuickTime Overview*. 2005 `http://developer.apple.com/documentation/QuickTime/RM/Fundamentals/QTOverview/QTOverview_Document/chapter_1000_section_1.html#//apple_ref/doc/uid/TP30000992-CH1g-QuickTimeOverview`

[App08] APPLE, Inc.: *3D Graphics Programming with QuickDraw 3D 1.5.4*. `http://developer.apple.com/documentation/QuickTime/QD3D/qd3d_book.htm`, Juli 2008

[BBD+02] BABCOCK, Brian ; BABU, Shivnath ; DATAR, Mayur ; MOTWANI, Rajeev ; WIDOM, Jennifer: Models and issues in data stream systems. In: *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA : ACM Press, 2002, S. 1–16

[BDG+04] BAUER, Martin ; DÜRR, Frank ; GEIGER, Jan ; GROSSMANN, Matthias ; HÖNLE, Nicola ; JOSWIG, Jean ; NICKLAS, Daniela ; SCHWARZ, Thomas: Information Management and Exchange in the Nexus Plaform / Universität Stuttgart. 2004. – Forschungsbericht. – Technical Report No. 2004/0

[BJRW03] BAUER, M. ; JENDOUBI, L. ; ROTHERMEL, K. ; WESTKÄMPER, E.: Grundlagen ubiquitärer Systeme und deren Anwendung in der „Smart Factory“. In: *Gronau, Norbert (ed.); Krallmann, Hermann; Scholz-Reiter, Bernd (ed.): Industrie Management - Zeitschrift für industrielle Geschäftsprozesse.* 19 (2003)

[Che04] CHERRY, S.: Edholm's law of bandwidth. In: *Spectrum, IEEE* 41 (Juli 2004), Nr. 7, S. 58–60

[CLRS04] CORMEN, Th. H. ; LEISERSON, C. E. ; RIVEST, R. ; STEIN, C.: *Algorithmen - Eine Einführung*. Oldenbourg Wissenschaftsverlag GmbH, 2004

[Con08] CONSORTIUM, Web3D: *Xj3D-Project*. `http://www.xj3d.org/`, Juli 2008

[Coo08] COOPERATION, E-Ink: *Low power, electronic paper displays*. `http://www.eink.com/index.html`, Juni 2008

[Del34] DELAUNAY, Boris N.: Sur la sphère vide. In: *Bulletin of Academy of Sciences of the USSR* (1934), Nr. 6, S. 793–800

[FB96] FREED, N. ; BORENSTEIN, N.: *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. United States, 1996

[For86] FORTUNE, S.: A sweepline algorithm for Voronoi diagrams. In: *SCG '86: Proceedings of the second annual symposium on Computational geometry*. New York, NY, USA : ACM, 1986. – ISBN 0–89791–194–6, S. 313–322

[Fos02] FOSTER, Ian: What is the Grid? - a three point checklist. In: *GRIDtoday* 1 (2002), Juli, Nr. 6. `http://www.gridtoday.com/02/0722/100136.html`

[FR94] FOLEY, J. ; RIBARSKY, B.: Next-generation Data Visualization Tools. In: *Scientific Visualization: Advances and Challenges* (1994)

[Fre08] FREEDESKTOP.ORG: *Xgl*. `http://www.freedesktop.org/wiki/Software/Xgl`, Juli 2008

[GML01] *Open GIS Consortium:Geography Markup Language (GML) Implementation Specification. Version 2.0,*. `http://www.opengis.net/gml/01-029/GML2.html`, 2001

[GSJ+06] GE, Jinghua ; SANDIN, Daniel J. ; JOHNSON, Andrew ; PETERKA, Tom ; KOOIMA, Robert ; GIRADO, Javier I. ; DEFANTI, Thomas A.: Point-based VR visualization for large-scale mesh datasets by real-time remote computation. In: *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*. New York, NY, USA : ACM, 2006, S. 43–50

[IEE07] IEEE: *IEEE Standard for Information technology — Telecommunications and information exchange between systems — Local and metropolitan area networks — Specific requirements*. `http://standards.ieee.org/getieee802/download/802.11-2007.pdf`, 2007. – Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications

[IET03] IETF: *RTP: A Transport Protocol for Real-Time Applications*. `http://www.ietf.org/rfc/rfc3550.txt`, Juli 2003. – RFC 3550

[ISO04a] ISO/IEC: *14496-2: Information technology — Coding of audio-visual objects —*. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40742, 2004. – Part 2: Visual

[ISO04b] ISO/IEC: *ISO/IEC 19775: Information technology — Computer graphics and image processing — Extensible 3D (X3D)*. http://www.web3d.org/x3d/specifications/x3d_specification.html, 2004

[IT06] ITU-T: *SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS Infrastructure of audiovisual services — Systems and terminal equipment for audiovisual services — Packet-based multimedia communications systems*. http://www.itu.int/rec/T-REC-H.323/en/, Juni 2006. – Recommendation

[KE01] KEMPER, A. ; EICKLER, A.: *Datenbanksysteme – Eine Einführung*. 4., aktualisierte und erweiterte. München, Wien : R. Oldenbourg Verlag, 2001

[KRHP04] KRANZLMULLER, Dieter ; ROSMANITH, Herbert ; HEINZLREITER, Paul ; POLAK, Martin: Interactive Virtual Reality on the Grid. In: *DS-RT '04: Proceedings of the 8th IEEE International Symposium on Distributed Simulation and Real-Time Applications*. Washington, DC, USA : IEEE Computer Society, 2004, S. 152–158

[Law72] LAWSON, C. L.: *Transforming Triangulations*. Discrete Mathematics, 1972. – 365–372 S.

[Lin93] LINN, J.: *RFC 1421: Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*. Februar 1993. – Status: PROPOSED STANDARD.

[Mat03] MATTERN, F.: Vom Verschwinden des Computers – Die Vision des Ubiquitous Computing. In: *Total Vernetzt*, Springer-Verlag, 2003, S. 1–41

[McC87] MCCORMICK, B. H.: Visualization in scientific computing. In: *Computer Graphics* 21 (1987), Nr. 6, S. 1–14

[Mic08] MICROSOFT: *Direct3D*. http://msdn.microsoft.com/de-de/library/bb205066(en-us).aspx, Juli 2008

[Nie93] NIELSEN, Jakob: *Usability Engineering*. 1st edition. Morgan Kaufmann, 1993

[NM01] NICKLAS, Daniela ; MITSCHANG, Bernhard: The Nexus Augmented World Model: An Extensible Approach for Mobile, Spatially-Aware Applications. In: *Wang, Yingxu (ed.); Patel, Shushma (ed.); Johnston, Ronald (ed.): Proceedings of the 7th International Conference on Object-Oriented Information Systems : OOIS*, 2001, S. 392–401

[SA06] SEGAL, Mark ; AKELEY, Kurt: *The OpenGL Graphics System: A Specification*. 2006

[Sar08] SARDINA, Daniel G.: *Framework for Distributed Data Processing*, Universität Stuttgart, Diplomarbeit, 2008. – Master Thesis No. 2785

[SB03] SHALF, John ; BETHEL, E. W.: How the Grid Will Affect the Architecture of Future Visualization Systems. In: *IEEE Computer Graphics and Applications* 23 (2003), May/June, Nr. 2, 6–9. `http://vis.lbl.gov/Publications/2002/IEEECGA-VisViewpointsMay2003-LBNL-51723.pdf`

[SHB+03] STEPANOV, Illya ; HÄHNER, Jörg ; BECKER, Christian ; TIAN, Jing ; ROTHERMEL, Kurt: A Meta-Model and Framework for User Mobility in Mobile Networks. In: *Proceedings of the 11th International Conference on Networking 2003 (ICON 2003), Sydney, Australia, September 28 - October 1, 2003*, IEEE, September 2003, 231–238

[She68] SHEPARD, Donald: A two-dimensional interpolation function for irregularly-spaced data. In: *Proceedings of the 1968 23rd ACM national conference*. New York, NY, USA : ACM, 1968, S. 517–524

[SHGN04] SCHWARZ, Thomas ; HÖNLE, Nicola ; GROSSMANN, Matthias ; NICKLAS, Daniela: A Library for Managing Spatial Context Using Arbitrary Coordinate Systems. In: *PerCom Workshops*, 2004, S. 48–54

[SJR+04] SINGH, R. ; JEONG, Byungil ; RENAMBOT, L. ; JOHNSON, A. ; LEIGH, J.: TeraVision: a distributed, scalable, high resolution graphics streaming system. In: *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*. Washington, DC, USA : IEEE Computer Society, 2004, S. 391–400

[SM99] SUN MICROSYSTEMS, Inc.: *Java (TM) Media Framework API Guide*. 1999

[SM00] SCHUMANN, Heidrun ; MÜLLER, Wolfgang: *Visualisierung - Grundlagen und allgemeine Methoden*. Springer Verlag, 2000

[SM08a] SUN MICROSYSTEMS, Inc.: *The AWT in 1.0 and 1.1*. `http://java.sun.com/products/jdk/awt/`, Juli 2008

[SM08b] SUN MICROSYSTEMS, Inc.: *Interface BlockingQueue<E>*. `http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/BlockingQueue.html`, Juni 2008

[SM08c] SUN MICROSYSTEMS, Inc.: *Java Native Interface Specification*. `http://java.sun.com/javase/6/docs/technotes/guides/jni/spec/jniTOC.html`, Juni 2008

[Sny87] SNYDER, J.P.: *Map Projections—A Working Manual*. US Geological Survey Professional Paper 1395. US Government Printing Office, Washington, DC, 1987

[SRD00] SOWIZRAL, Henry ; RUSHFORTH, Kevin ; DEERING, Michael: *The Java 3D(TM) API Specification*. 2nd Edition. Sun Microsystems, Inc., 2000

[Tan03] TANENBAUM, Andrew S.: *Computer networks*. 4th edition. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 2003. – ISBN 0–13–162959–X

[TMSF03] TUCKER, Peter A. ; MAIER, David ; SHEARD, Tim ; FEGARAS, Leonidas: Exploiting Punctuation Semantics in Continuous Data Streams. In: *IEEE Trans. on Knowl. and Data Eng.* 15 (2003), Nr. 3, S. 555–568

[Ull07] ULLENBOOM, Christian: *Java ist auch eine Insel*. 7., aktualisierte Auflage. Bonn : Galileo Computing, 2007 `http://www.galileocomputing.de/openbook/javainsel7/`

[W3C04] W3C: *XML Schema Part 1: Structures Second Edition*. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html`, oct 2004. – W3C Recommendation

[W3C07] W3C: *XML Path Language (XPath) 2.0*. `http://www.w3.org/TR/xpath20/`, Januar 2007. – Recommendation

[WBW96] WOOD, Jason ; BRODLIE, Ken ; WRIGHT, Helen: Visualization over the World Wide Web and its application to environmental data. In: *VIS '96: Proceedings of the 7th conference on Visualization '96*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1996, S. 81–ff.

[Zep04] ZEPPENFELD, Klaus: *Lehrbuch der Grafikprogrammierung*. 1. Auflage. Elsevier GmbH München, 2004

Alle URLs wurden zuletzt am 12.07.2008 geprüft.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Carlos Lübbe)